Processes

Operating Systems

G53OPS: Operating Systems

Graham Kendall

Processes

Operating Systems

- Concept of a process is fundamental to an operating system
- Can be viewed as an abstraction of a program
  - Although to be strict we can say that a program (i.e. an algorithm expressed in some suitable notation) has a process that executes the algorithm and has associated with it input, output and a state
- Computers nowadays can do many things at the same time. They can be writing to a printer, reading from a disc and scanning an image

Processes

Operating Systems

- The computer (more strictly the operating system) is also responsible for running many processes, usually, on the same CPU
- Must give the illusion that the computer is doing many things at the same time (pseudoparallelism)

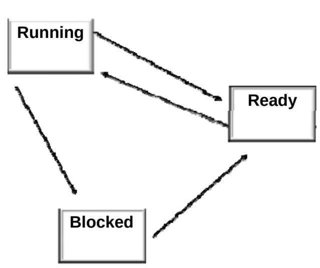
Processes

Operating Systems

- Important to realise
  - The CPU is switching processes
  - One process can have an effect on another process which is not currently running

Processes

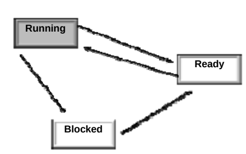
Operating Systems



Processes

Operating Systems

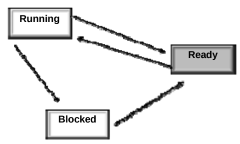
- **Running**. Only one process can be running at any one time (assuming a single processor machine). A running process is the process that is actually using the CPU at that time



Processes

Operating Systems

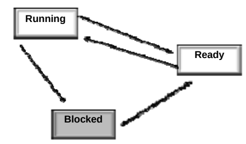
- **Ready**. A process that is ready is runnable but cannot get access to the CPU due to another process using it



Processes

Operating Systems

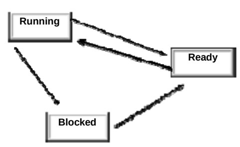
- **Blocked**. A blocked process is unable to run until some external event has taken place. For example, it may be waiting for data to be retrieved from a disc



Processes

Operating Systems

- The scheduler is concerned with deciding which one of the processes in a ready state should be allowed to move to a running state



Processes

Operating Systems

- When a process moves from a running state to a ready or blocked state it must store certain information so that it can restart from the same point when it moves back to a running state
  - Which instruction it was about to execute
  - Which record it was about to read from its input file
  - Values in the registers

Processes

Operating Systems

Each process has a process table

Process Management
Registers
Program counter
Program status word
Stack pointer
Process State
Time when process started
CPU time used
Time of next alarm
Process id

Note that accounting information is stored as well

Processes

Operating Systems

- Sometimes necessary for two processes to communicate with one another
- Not a situation where a process can write some data to a file that is read by another process at a later time
- e.g. One type of process (i.e. there could be more than one process of this type running) that checks a counter when it starts running. If the counter is at a certain value, say *x*, then the process terminates as only *x* copies of the process are allowed to run at any one time

Processes

- This is how it might work
  - The process starts
  - The counter, *i*, is read from the shared memory
  - If the $i = x$ the process terminates else $i = i + 1$
  - *x* is written back to the shared memory

Processes

- But consider this scenario
  - Process 1, P_1, starts
  - P_1 reads the counter, i_1, from the shared memory. Assume $i_1 = 3$ (that is three processes of this type are already running)
  - P_1 gets interrupted and is placed in a ready state
  - Process 2, P_2, starts
  - P_2 reads the counter, i_2, from the shared memory; $i_2 = 3$
  - Assume $i_2 < x$ so $i_2 = i_2 +1$ (i.e. 4)
  - i_2 is written back to shared memory
  - P_2 is moved to a ready state and P_1 goes into a running state
  - Assume $i_1 < x$ so $i_1 = i_1 +1$ (i.e. 4)
  - i_1 is written back to the shared memory

Processes

- Five processes running but the counter is only set to four
- This problem is known as a **race condition**

Processes

- Avoid race conditions by not allowing two processes to be in their critical sections at the same time
- We need a mechanism of mutual exclusion
- Some way of ensuring that one processes, whilst using the shared variable, does not allow another process to access that variable

Processes

- In fact we need four conditions to hold
  1. No two processes may be simultaneously inside their critical sections
  2. No assumptions may be made about the speed or the number of processors
  3. No process running outside its critical section may block other processes
  4. No process should have to wait forever to enter its critical section
- It is difficult to devise a method that meets all these conditions, but let's try….

Processes

- **Disabling Interrupts**
  - Allow a process to disable interrupts before it enters its critical section and then enable interrupts after it leaves its critical section
  - CPU will be unable to switch processes
  - Guarantees that the process can use the shared variable without another process accessing it
  - But, disabling interrupts, is a major undertaking
  - At best, the computer will not be able to service interrupts for, maybe, a long time
  - At worst, the process may never enable interrupts, thus (effectively) crashing the computer
  - The disadvantages far outweigh the advantages

Processes

Operating Systems

- **Lock Variables**
  - Another method is to assign a lock variable
  - A process is only able to enter a critical section when the lock variable is set to zero
  - A process sets the variable to (say) 1 when a process is in its critical section and resets it to zero when a processes exits its critical section
  - Does this work okay?
  - But this is flawed as it simply moves the problem from the shared variable to the lock variable

Processes

Operating Systems

- **Strict Alternation**

Process 0

```
While (TRUE) {
        while (turn != 0); // wait
        critical_section();
        turn = 1;
        noncritical_section();
}
```

Process 1

```
While (TRUE) {
        while (turn != 1); // wait
        critical_section();
        turn = 0;
        noncritical_section();
}
```


Processes

Operating Systems

- **Strict Alternation**

Process 0

```
While (TRUE) {
        while (turn != 0);
        critical_section();
        turn = 1;
        noncritical_section();
}
```

Process 1

```
While (TRUE) {
        while (turn != 1);
        critical_section();
        turn = 0;
        noncritical_section();
}
```

turn = 0

<u>Process 0</u>	<u>Process 1</u>
while(turn != 0) → F	while(turn != 1) → T
critical section()	
turn = 1	
noncritical_section()	critical section()
	turn = 0

Processes

Operating Systems

- **Strict Alternation**

Process 0

```
While (TRUE) {
        while (turn != 0); // wait
        critical_section();
        turn = 1;
        noncritical_section();
}
```

Process 1

```
While (TRUE) {
        while (turn != 1); // wait
        critical_section();
        turn = 0;
        noncritical_section();
}
```

- **Assume the variable turn is initially set to zero.**
- **Process 0 runs. And finding turn is zero, it enters its critical region**
- **If process 1 tries to run, it will find that turn is zero and will have to wait**
- **When process 0 exits its critical region turn = 1 and process 1 can continue**
- **Process 0 is now blocked**

Processes

Operating Systems

- **Strict Alternation**

Process 0

```
While (TRUE) {
        while (turn != 0); // wait
        critical_section();
        turn = 1;
        noncritical_section();
}
```

Process 1

```
While (TRUE) {
        while (turn != 1); // wait
        critical_section();
        turn = 0;
        noncritical_section();
}
```

What is the problem with this approach?

Processes

Operating Systems

- **Strict Alternation**

Process 0

```
While (TRUE) {
        while (turn != 0); //
wait
        critical_section();
        turn = 1;
        noncritical_section();
}
```

Process 1

```
While (TRUE) {
        while (turn != 1); //
wait
        critical_section();
        turn = 0;
        noncritical_section();
}
```

- Process 0 runs, enters its critical section and exits; setting turn to 1. Process 0 is now in its non-critical section. Assume this non-critical procedure takes a long time.
- Process 1, which is a much faster process, now runs and once it has left its critical section turn is set to zero.
- Process 1 executes its non-critical section very quickly and returns to the top of the procedure.
- The situation is now that process 0 is in its non-critical section and process 1 is waiting for turn to be set to zero. In fact, there is no reason why process 1 cannot enter its critical region as process 0 is not in its critical region.

Processes

Operating Systems

- **Strict Alternation**

Process 0

```
While (TRUE) {
            while (turn != 0); // wait
            critical_section();
            turn = 1;
            noncritical_section();
}
```

Process 1

```
While (TRUE) {
            while (turn != 1); // wait
            critical_section();
            turn = 0;
            noncritical_section();
}
```

- We have a violation of one of the conditions that we listed above
  1. No two processes may be simultaneously inside their critical sections
  2. No assumptions may be made about the speed or the number of processors
  3. **No process running outside its critical section may block other processes**
  4. No process should have to wait forever to enter its critical section

Processes

Operating Systems

- **Peterson's Solution**

- Solution to the mutual exclusion problem that does not require strict alternation
- Still uses the idea of lock (and warning) variables together with the concept of taking turns

Processes

Operating Systems

```
int No_Of_Processes;
int turn;
int interested[No_Of_Processes];

void enter_region(int process) {
     int other;
     other = 1 – process;
     interested[process] = TRUE;
     turn = process;
     while(turn == process &&
             interested[other] == TRUE);
}
void leave_region(int process) {
     interested[process] = FALSE;
}
```


Processes

Operating Systems

- **Peterson's Solution**

```
int No_Of_Processes;
int turn;
int interested[No_Of_Processes];

void enter_region(int process) {
      int other;
      other = 1 – process;
      interested[process] = TRUE;
      turn = process;
      while(turn == process &&
      interested[other] == TRUE);
}
void leave_region(int process) {
      interested[process] = FALSE;
}
```

- Using Peterson's algorithm work out what will happen, given the following sequence. Assume that we are only interested in controlling two processes.
- A process, P_0, starts and calls enter_region. Assume no other processes are running
- Once P_0 is in its critical region what happens if another process, P_1, starts and calls enter_region?
- P_0 calls leave_region

Processes

Operating Systems

Process 0

```
int No_Of_Processes;
int turn;
int interested[No_Of_Processes];

void enter_region(int process) {
       int other;
       other = 1 – process;
       interested[process] = TRUE;
       turn = process;
       while(turn == process && interested[other] == TRUE);
}
```

```
No_Of_Processes 2;
int turn;
int interested[2];

enter_region(0) {
       int other;
       other = 1;
       interested[0] = TRUE;
       turn = 0;
       while(0 == 0 && FALSE == TRUE);
}
```


Processes

Operating Systems

- **Peterson's Solution**
- Initially, the array *interested* has all (both) its elements set to false
- Assume process 0 calls enter_region. The variable *other* is set to one (the other process number) and it indicates its interest by setting the relevant element of interested, and sets the turn variable

Turn == Process	Interested [0]	Action
F	F	Continue
F	T	Continue
T	F	Continue
T	T	Wait

```
No_Of_Processes 2;
int turn;
int interested[2];

enter_region(0) {
       int other;
       other = 1;
       interested[0] = TRUE;
       turn = 0;
       while(0 == 0 && FALSE == TRUE);
}
```

- In this instance, the process will be allowed to enter its critical region, as process 1 is not interested in running

Race Conditions

Processes

Operating Systems

Process 1

```
int No_Of_Processes;
int turn;
int interested[No_Of_Processes];

void enter_region(int process) {
        int other;
        other = 1 – process;
        interested[process] = TRUE;
        turn = process;
        while(turn == process && interested[other] == TRUE);
}
```

```
No_Of_Processes 2;
int turn;
int interested[2];

enter_region(1) {
        int other;
        other = 0;
        interested[1] = TRUE;
        turn = 1;
        while(1 == 1 && TRUE == TRUE);
}
```


Race Conditions

Processes

Operating Systems

- **Peterson's Solution**
  - Now process 1 could call enter_region. It will be forced to wait as the other process (0) is still interested. Process 1 will only be allowed to continue when interested[0] is set to false which can only come about from process 0 calling leave_region

Turn == Process	Interested [1]	Action
F	F	Continue
F	T	Continue
T	F	Continue
T	T	Wait

```
No_Of_Processes 2;
int turn;
int interested[2];

enter_region(1) {
        int other;
        other = 0;
        interested[1] = TRUE;
        turn = 1;
        while(1 == 1 && TRUE == TRUE);
}
```


Race Conditions

Processes

Operating Systems

- **Peterson's Solution**

- **What happens if two processes call enter_region at exactly the same time?**

Race Conditions

Processes

Operating Systems

```
void enter_region(int process) {
        int other;
        other = 1 – process;
        interested[process] = TRUE;
        turn = process;
         while(turn == process && interested[other] == TRUE);
}
```

Process 0

```
No_Of_Processes 2;
int turn;
int interested[2];

enter_region(0) {
        int other;
        other = 1;
        interested[0] = TRUE;
        turn = 0;
- - - - - - - - - - - - - - - - - - - -
        while …;
}
```

Time ↓

Process 1

```
No_Of_Processes 2;
int turn;
int interested[2];

enter_region(1) {
        int other;
        other = 0;
        interested[1] = TRUE;
- - - - - - - - - - - - - - - - - - - -
        turn = 1;
        while …;
}
```

Process 0 sets turn, but it is immedietely overwritten by Process 1

Race Conditions

Processes

Operating Systems

- **Peterson's Solution**
  - Assume that process 0 sets turn to zero and then process 1 immediately sets it to 1. Under these conditions process 0 will be allowed to enter its critical region and process 1 will be forced to wait

- Turn (=1) == Process (=0) → F
- Interested[0] → T

Turn == Process	Interested[0]	Action
F	F	Continue
F	T	Continue
T	F	Continue
T	T	Wait

Process 0

- Turn (=1) == Process (=1) → T
- Interested[1] → T

Turn == Process	Interested[1]	Action
F	F	Continue
F	T	Continue
T	F	Continue
T	T	Wait

Process 1

Race Conditions

Processes

Operating Systems

- **Test and Set Lock**
- Some instruction sets assist us in implementing mutual exclusion
- The instruction is commonly called Test and Set Lock (TSL)
- It reads the contents of a memory location, stores it in a register and then stores a non-zero value at the address.
- Guaranteed to be indivisible

Race Conditions

Processes

Operating Systems

• **Test and Set Lock**

Solution to mutual exclusion using assembly code

```
enter_region:
      tsl        register, flag    ; copy flag to register and set
                                   flag to 1
      cmp        register, #0      ;was flag zero?
      jnz        enter_region      ;if flag was non zero, lock was
                                   set, so loop
      ret                          ;return (and enter critical
                                           region)

leave_region:
      mov        flag, #0           ;store zero in flag
      ret                           ;return
```

Race Conditions

Processes

Operating Systems

• **Test and Set Lock**

- Assume, two processes.
- Process 0 calls enter_region
- TSL copies the flag to a register and sets it to a non-zero value
- The flag is compared to zero and if found to be non-zero the routine loops back to the top
- Only when process 1 has set the flag to zero (or under initial conditions) will process 0 be allowed to continue

```
enter_region:
      tsl        register, flag
      cmp        register, #0
      jnz        enter_region
      ret

leave_region:
      mov        flag
      ret
```

Race Conditions

Processes

Operating Systems

- Peterson's Solution and TSL both solve the mutual exclusion problem
- However, both of these solutions sit in a tight loop waiting for a condition to be met (busy waiting).
- Wasteful of CPU resources
- Any other problems with these approaches?

Race Conditions

Processes

Operating Systems

Other disadvantages

- Suppose we have two processes, one of high priority and one of low priority
- The scheduling algorithm runs the high priority process whenever it is in ready state
- If the low priority process is in its critical section when the high priority process becomes ready to run the low priority process will be placed in a ready state so that the higher priority process can be run
- But, the high priority process will not be able to run and the low priority process cannot run again to release the higher priority process
- This is sometimes known as the priority inversion problem

Race Conditions

Processes

Operating Systems

Sleep/Wakeup

- **Sleep()**: System call that causes the calling process to block until woken up
- **Wakeup(*process*)**: Causes a sleeping process to wakeup (i.e. become available to run)

Race Conditions

Processes

Operating Systems

Classic Synchronisation Problems

Producer/Consumer Problem

- A producer process generates information that is to be processed by the consumer process
- The processes can run concurrently through the use of a buffer
- The consumer must wait on an empty buffer
- The producer must wait on a full buffer

Operating Systems

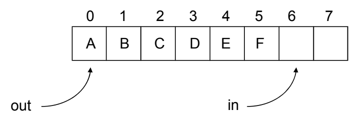
Race Conditions

Processes

Classic Synchronisation Problems

Producer/Consumer Problem

Also known as the bounded buffer problem



Operating Systems

Race Conditions

Processes

Producer/Consumer Problem

sleep() and wakeup() are not part of C standard library, but let's assume that they are available

Insert_item, remove_item and consume_item do as their name suggests, along with relevant housekeeping

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                           /* repeat forever */
        item = produce_item( );              /* generate next item */
        if (count == N) sleep( );            /* if buffer is full, go to sleep */
        insert_item(item);                   /* put item in buffer */
        count = count + 1;                   /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                           /* repeat forever */
        if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
        item = remove_item( );               /* take item out of buffer */
        count = count - 1;                   /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                  /* print item */
    }
}
```

Operating Systems

Race Conditions

Processes

Producer/Consumer Problem

Does this model the problem correctly?

If not, why not?


```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                           /* repeat forever */
        item = produce_item( );              /* generate next item */
        if (count == N) sleep( );            /* if buffer is full, go to sleep */
        insert_item(item);                   /* put item in buffer */
        count = count + 1;                   /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                           /* repeat forever */
        if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
        item = remove_item( );               /* take item out of buffer */
        count = count - 1;                   /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                  /* print item */
    }
}
```


Operating Systems

Race Conditions

Processes

Producer/Consumer Problem

Imagine this scenario

- The buffer is empty
- Consumer has just read count (=0)
- Consumer gets pre-empted and calls the producer
- The producer inserts an item into the buffer, increments counts and notes count=1
- Reasoning that count was just zero (and that the consumer was sleeping) it sends a wakeup
- Consumer is not actually asleep, so the wakeup is lost
- When consumer next runs, it will go to sleep
- Eventually producer will also sleep, and both will sleep forever

Operating Systems

Race Conditions

Processes

Producer/Consumer Problem

The problem

- The problem is that the wakeup request was lost as the other process was not sleeping.
- We could solve (for this simple case) with a wakeup flag so that if a process subsequently sleeps, and the wakeup flag is set, it would wakeup immediately (or not even sleep)
- But we can devise more complicated scenarios where this solution becomes impractical.

Operating Systems

Race Conditions

Processes

Semaphores

- Use an integer varaiable to count the number stored wakeups
- *semaphore* = 0 indcates that no wakeups are stored
- *semaphore* > 0 indcates that some wakeups are pending
- `down`/`up` propsed as generalisations of `sleep`/`wakeup`

Race Conditions

Processes

Operating Systems

Semaphores

down
```
if(semaphore > 0) {
     semaphore--;
     // and continue
else
     sleep();
     // when woken
     semaphore--;
}
```

up
```
semaphore++;
Choose process (at random?) that is sleeping on
  semaphore and allow it to continue
```

Race Conditions

Processes

Operating Systems

Semaphores

- down must be a single, indivisible atomic action. Once started, no other process can access the semaphore until completed or blocked (sleeping)

down
```
if(semaphore > 0) {
     semaphore--;
     // and continue
else
     sleep();
     // when woken
     semaphore--;
}
```

up
```
semaphore++;
Choose process (at random?) that is sleeping on semaphore and allow it to
  continue
```

- up must be a single, indivisible atomic action

Race Conditions

Processes

Operating Systems

Producer/Consumer Problem

Three semaphores

- *mutex*, to ensure that the producer/consumer do not access the buffer at the same time
- *empty*, for counting the number of slots that are empty
- *full*, for counting the number of slots that are full

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item();       /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

Race Conditions

Processes

Operating Systems

Producer/Consumer Problem

Mutex
- Initially 1
- Ensures that only one process can be in its critical region
- A **binary semaphore** as it can only take one of two values


```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item();       /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

Race Conditions

Processes

Operating Systems

Sequence of calls (mutex = 1)

- $Producer_1$ (mutex = 0)
- $Producer_2$ (process sleeps)
- $Producer_1$ ends (mutex++, $Producer_2$ is woken, mutex--; mutex=0)
- *If another producer was now invoked, it would sleep*
- $Producer_2$ ends (mutex = 1)

down
```
if(mutex > 0) {
     mutex--;
     // and continue
else
     sleep();
     // when woken
     mutex--;
}
```

up
```
mutex++;
Choose process (at random?) that
  is sleeping on mutex and allow
  it to continue
```

```
down(&mutex)

insert_item(item)

up(&mutex)
```

Race Conditions

Processes

Operating Systems


```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item();       /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

Race Conditions

Processes

Operating Systems

Produce Item (empty = 100, full = 0)

~~item = produce_item();~~
down(&empty);
~~down(&mutex);~~
insert_item(item);
~~up(&mutex);~~
up(&full);

Consume Item (empty = 100, full = 0)

down(&full);
~~down(&mutex);~~
item = remove_item();
~~up(&mutex);~~
up(&empty);
~~consume_item(item);~~

Race Conditions

Processes

Operating Systems

Produce Item (empty = 100, full = 0)

- Will allow 100 items to be produced before sleeping [down(&*empty*)]
- Will increment the full mutex [up(&*full*)] (and let's assume nothing is sleeping at the moment)

down

```
if(empty > 0) {
        empty--;
        // and continue
else
        sleep();
        // when woken
        empty--;
}
```

up

```
full++;
Choose process (at random?) that
   is sleeping on full and allow
   it to continue
```

```
down(&empty);
insert_item(item);
up(&full);
```

Race Conditions

Processes

Operating Systems

Consume Item (empty = 100, full = 0)

- Will allow items to be consumed while items are available. If not available, then sleep [down(&*full*)]
- Will increment the empty mutex [up(&*empty*)]

down

```
if(full > 0) {
        full--;
        // and continue
else
        sleep();
        // when woken
        full--;
}
```

up

```
empty++;
Choose process (at random?) that
   is sleeping on empty and allow
   it to continue
```

```
down(&full);
item = remove_item();
up(&empty);
```

Race Conditions

Processes

Operating Systems

"Boundary" Condition

- Assume *full* = 100, *empty* = 0 (i.e. 100 items and been produced, with none consumed)

```
down(&empty);
insert_item(item);
up(&full);
```

- Calling producer again will send that consumer to sleep
- If we now call up(empty), via a consumer, we will wake up the producer

```
down(&full);
item = remove_item();
up(&empty);
```

- The same principles for the consumer trying to consume from an empty queue.

down (empty)

```
if(empty > 0) {
        empty--;
        // and continue
else
        sleep();
        // when woken
        empty--;
}
```

up (full)

```
full++;
Choose process (at random?) that is
   sleeping on full and allow it to
   continue
```

down (full)

```
if(full > 0) {
        full--;
        // and continue
else
        sleep();
        // when woken
        full--;
}
```

up (empty)

```
empty++;
Choose process (at random?) that is
   sleeping on empty and allow it to
   continue
```

Race Conditions

Processes

Operating Systems

Notes

We have used semaphores in two ways

- Mutual Exclusion (mutex in the example), to protect critical regions
- Syncronisation (empty/full) to protect the buffer from under/over-flowing

Scheduling

Processes

Operating Systems

Scheduling Objectives

- **Fairness** : Ensure each process gets a fair share of the CPU
- **Efficiency** : Ensure the CPU is busy 100% of the time. In practise, a measure of between 40% (for a lightly loaded system) to 90% (for a heavily loaded system) is acceptable
- **Response Times** : Ensure interactive users get good response times
- **Turnaround** : Ensure batch jobs are processed in acceptable time
- **Throughput** : Ensure a maximum number of jobs are processed

Cannot meet all of these objectives to an optimum (classic multi-objective problem)

Scheduling

Processes

Operating Systems

Non Pre-emptive Scheduling

- Allowing a process to run until it has completed has some advantages
  - We would no longer have to concern ourselves with race conditions as we could be sure that one process could not interrupt another and update a shared variable
  - Scheduling the next process to run would simply be a case of taking the highest priority job (or using some other algorithm, such as FIFO (First-in, First-out)

Scheduling

Processes

Operating Systems

Non Pre-emptive Scheduling

The disadvantages far outweigh the advantages.

- A rogue process may never relinquish control, effectively bringing the computer to a standstill
- Processes may hold the CPU too long, not allowing other applications to run

Scheduling

Processes

Operating Systems

Pre-emptive Scheduling

Tasks of the Scheduler

- To decide which process can use the CPU
- Once it has had a period of time then it is placed into a ready state and the next process allowed to run

This disadvantage of this method is that we need to cater for race conditions as well as having the responsibility of scheduling the processes

Scheduling

Processes

Operating Systems

Typical Process Activity

Typical processes come in two varieties

- I/O bound processes which require the CPU in short bursts
- Processes that require the CPU for long bursts

Scheduling

Processes

Operating Systems

Typical Process Activity

CPU Burst Time

- How long the process needs the CPU before it will either finish or move to a blocked state

We cannot know the burst time of a process before it runs

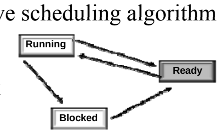
Scheduling

Processes

Operating Systems

First Come, First Served (FCFS)

- Execute processes in the order they arrive and execute them to completion
- This is simply a non-preemptive scheduling algorithm
- Easy to implement
  - Add Process Control Block to the ready queue



- Problem is that the average waiting time can be long

Scheduling

Processes

Operating Systems

First Come, First Served (FCFS)

Process	Burst Time	Wait Time
P_1	27	0
P_2	9	27
P_3	2	36

Average waiting time of 21ms ((0+27+36) /3)

Process	Burst Time	Wait Time
P_1	9	0
P_2	2	9
P_3	27	11

Average waiting time of 6.6ms ((0+9+11) /3)

Scheduling

Processes

Operating Systems

First Come, First Served (FCFS)

The FCFS algorithm can have undesirable effects.

- A CPU bound job may make the I/O bound (once they have finished the I/O) wait for the processor. At this point the I/O devices are sitting idle
- When the CPU bound job finally does some I/O, the mainly I/O bound processes use the CPU quickly and now the CPU sits idle waiting for the mainly CPU bound job to complete its I/O

Scheduling

Processes

Operating Systems

Shortest Job First (SJF)

- Each process is tagged with the length of its next CPU burst
- The processes are scheduled by selecting the shortest job first.

Scheduling

Processes

Operating Systems

Shortest Job First (SJF)

Process	Burst Time	Wait Time
P_1	12	0
P_2	19	12
P_3	4	31
P_4	7	35

FCFS: Average waiting time is 19.50ms (78/4)

Process	Burst Time	Wait Time
P_3	4	0
P_4	7	4
P_1	12	11
P_2	19	23

SJF: Average waiting time is 9.50ms (38/4)

Scheduling

Processes

Operating Systems

Shortest Job First (SJF)

- The SJF algorithm is provably optimal with regard to the average waiting time
- Therefore, we should always use this scheduling algorithm
- But, do you see any problems?

Scheduling

Processes

Operating Systems

Shortest Job First (SJF)

- The problem is we do not know the burst time of a process before it starts
- For some systems (notably batch systems) we can make fairly accurate estimates but for interactive processes it is not so easy

Processes

Shortest Job First (SJF)

- One approach is to try and estimate the length of the next CPU burst, based on the processes previous activity
- To do this we can use the following formula

$T_{n+1} = at_n + (1 - a)T_n$

where

a, $0 <= a <= 1$

T_n, stores the past history

t_n, contains the most recent information

Scheduling

Processes

Shortest Job First (SJF)

$T_{n+1} = at_n + (1 - a)T_n$

where
a, $0 <= a <= 1$
T_n, stores the past history
t_n, contains the most recent information

- This formula allows us to weight both the history of the burst times and the most recent burst time
- If $a = 0$ then $T_{n+1} = T_n$ and recent history (the most recent burst time) has no effect. If $a = 1$ then the history has no effect and the guess is equal to the most recent burst time
- A value of 0.5 for a is often used so that equal weight is given to recent and past history

See Spreadsheet (Exercises)

Operating Systems

Scheduling

Processes

Priority Scheduling

- SJF is a special case of priority scheduling
- We can use a number of different measures as priority

Operating Systems

Scheduling

Processes

Priority Scheduling

Example of priorities based on the resources they have previously

- Assume processes are allowed 100ms before the scheduler preempts it
- If a process used, say 2ms it is likely to be a job that is I/O bound
- It is in the schedulers interest to allow this job to run as soon as possible
- If a job uses all its 100ms we might give it a lower priority, in the belief that we can get smaller jobs completed first

Operating Systems

Scheduling

Processes

Priority Scheduling

We could use this formula to calculate priorities

$1 / (n / p)$

where

n, is the last CPU burst for that process

p, is the CPU time allowed for each process before it is preempted (100ms in our example)

Operating Systems

Scheduling

Processes

Priority Scheduling

- Plugging in some real figures we can assign priorities as follows

CPU Burst Last Time (n)	Processing Time Slice (p)	Priority Assigned
100	100	1
50	100	2
25	100	4
5	100	20
2	100	50
1	100	100

$1 / (n / p)$

- The process which had the shortest previous burst time has the higher priority

Operating Systems

Scheduling

Processes

Operating Systems

Priority Scheduling

Also set priorities externally

- During the day interactive jobs are given a high priority
- Batch jobs given high priority overnight

Another alternative is to allow users who pay more for their computer time to be given higher priority for their jobs.

Scheduling

Processes

Operating Systems

Priority Scheduling

Problems with priority scheduling

- Some processes may never run (indefinite blocking or starvation)

Possible Solution

- Introduce aging

Scheduling

Processes

Operating Systems

Round Robin Scheduling

- Processes held in a queue
- Scheduler takes the first job off the front of the queue and assigns it to the CPU (as FCFS)
- Unit of time called a quantum is defined
- When quantum time is reached the process is preempted and placed at the back of queue
- Average waiting time can be quite long

Scheduling

Processes

Operating Systems

Round Robin Scheduling

- Consider these processes. Assume all arrive at time zero and *quantum* = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

Calculate the average waiting time for RR and SJF

Scheduling

Processes

Operating Systems

Round Robin Scheduling

- Consider these processes. Assume all arrive at time zero and *quantum* = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

Time Step	P_1	P_2	P_3
1	0 (E)	4	4
2	3	4 (E)	7
3	6	4	7 (E)
4	E		
5	….		

SJF Average Wait Time = 3

Average Wait time = 17
(6+4+7)/3 = 5.66

Scheduling

Processes

Operating Systems

Round Robin Scheduling

Main concern with the RR algorithm is the length of the quantum

- Too long and we have FCFS
- Switch processes every ms and we make it appear as if every process has its own processor that runs at 1/n the speed of the actual processor (ignoring overheads)

Example

- Set the quantum to 5ms and assume it takes 5ms to execute a process switch
- We are using half the CPU capability simply switching processes

Processes

Multilevel Queue Scheduling

- Two typical processes in a system
  - Interactive jobs – tend to be shorter
  - Batch jobs – tend to be longer
- Set up different queues to cater for different process types
- Each queue may have its own scheduling algorithm
- Background queue will typically use the FCFS algorithm
- Interactive queue may use the RR algorithm

Processes

Multilevel Queue Scheduling

- Scheduler has to decide which queue to run
- Two main methods
  - Higher priority queues can be processed until they are empty before the lower priority queues are executed
  - Each queue can be given a certain amount of the CPU
- Can be other queues
  - System queue – high priority

Processes

Multilevel Queue Scheduling

- Multilevel Queue Scheduling assigns a process to a queue and it remains in that queue
- May be advantageous to move processes between queues (multilevel feedback queue scheduling)
- Consider processes with different CPU burst characteristics
  - Process which use too much of the CPU will be moved to a lower priority queue
  - Leave I/O bound and (fast) interactive processes in the higher priority queue(s)

Processes

Multilevel Queue Scheduling

- Assume three queues (Q_0, Q_1 and Q_2)
  - Scheduler executes Q_0 and only considers Q_1 and Q_2 when Q_0 is empty
  - A Q_1 process is preempted if a Q_0 process arrives
  - New jobs are placed in Q_0
  - Q_0 runs with a quantum of 8ms
  - If a process is preempted it is placed at the end of the Q_1 queue
  - Q_1 has a time quantum of 16ms associated with it
  - Any processes preempted in Q_1 are moved to Q_2, which is FCFS

Processes

Multilevel Queue Scheduling

- Any jobs that require less than 8ms of the CPU are serviced *very* quickly
- Any processes that require between 8ms and 24ms are also serviced *fairly* quickly
- Any jobs that need more than 24ms are executed with any spare CPU capacity once Q_0 and Q_1 processes have been serviced

Processes

Multilevel Queue Scheduling

- Parameters that define the scheduler
  - The number of queues
  - The scheduling algorithm for each queue
  - The algorithm used to demote processes to lower priority queues
  - The algorithm used to promote processes to a higher priority queue (some form of aging)
  - The algorithm used to determine which queue a process will enter

Processes

Multilevel Queue Scheduling

- Mimic other scheduling algorithms
  - One queue
  - Suitable quantum
  - RR algorithm
  - Generalise to the RR algorithm

Processes

Multilevel Queue Scheduling

- Assumed that the processes are all available in memory so that the context switching is fast
- If the computer is low on memory then some processes may be swapped out to disc
- Context switching takes longer
- Sensible to schedule only those processes in memory
  - Responsibility of a top level scheduler

Processes

Multilevel Queue Scheduling

- Second scheduler is invoked periodically to remove processes from memory to disc and vice versa
- Parameters to decide which processes to move
  - How long has it been since a process has been swapped in or out?
  - How much CPU time has the process recently had?
  - How big is the process (on the basis that small ones do not get in the way)?
  - What is the priority of the process?

Processes

Evaluation of Scheduling Algorithms

- Not covered in (Tanenbaum, 1992) - In (Silberschatz, 1994)
- How do we decide which scheduling algorithm to use?
- How do we evaluate?
  - Fairness
  - Efficiency
  - Response Times
  - Turnaround
  - Throughput

Processes

Evaluation of Scheduling Algorithms

Deterministic Modeling

- Takes a predetermined workload and evaluates each algorithm
- Advantages
  - It is exact
  - It is fast to compute
- Disadvantages
  - Only applicable to the workload that you use to test

Processes

Evaluation of Scheduling Algorithms

- Given this workload, and assuming that all processes arrive at time zero
- Which of the following algorithms will perform best?
  - First Come First Served (FCFS)
  - Non Preemptive Shortest Job First (SJF)
  - Round Robin (RR)
- Assume a quantum of 8 milliseconds

Process	Burst Time
P_1	9
P_2	33
P_3	2
P_4	5
P_5	14

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

- Given this workload, and assuming that all processes arrive at time zero
- Which of the following algorithms will perform best?
  - First Come First Served (FCFS)
  - Non Preemptive Shortest Job First (SJF)
  - Round Robin (RR)
- Assume a quantum of 8 milliseconds

Process	Burst Time
P_1	9
P_2	33
P_3	2
P_4	5
P_5	14

SJF: 55/5 = 11.0
RR: 119/5 = 23.8
FCFS: 144/5 = 28.8

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

- Given this workload, and assuming that all processes arrive at time zero
- Which of the following algorithms will perform best?
  - First Come First Served (FCFS)
  - Non Preemptive Shortest Job First (SJF)
  - Round Robin (RR)
- Assume a quantum of 8 milliseconds

Process	Burst Time
P_1	8
P_2	33
P_3	2
P_4	5
P_5	14

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

- Given this workload, and assuming that all processes arrive at time zero
- Which of the following algorithms will perform best?
  - First Come First Served (FCFS)
  - Non Preemptive Shortest Job First (SJF)
  - Round Robin (RR)
- Assume a quantum of 8 milliseconds

Process	Burst Time
P_1	8
P_2	33
P_3	2
P_4	5
P_5	14

SJF: 53/5 = 10.6
RR: 94/5 = 18.8
FCFS: 140/5 = 28.0

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

- Given this workload, and assuming that all processes arrive at time zero
- Which of the following algorithms will perform best?
  - First Come First Served (FCFS)
  - Non Preemptive Shortest Job First (SJF)
  - Round Robin (RR)
- Assume a quantum of 15 milliseconds

Process	Burst Time
P_1	9
P_2	33
P_3	2
P_4	5
P_5	14

SJF: 55/5 = 11.0
RR: 111/5 = 22.2
FCFS: 144/5 = 28.8

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Process	Burst Time
P_1	9
P_2	33
P_3	2
P_4	5
P_5	14

SJF: 55/5 = 11.0
RR: 119/5 = 23.8
FCFS: 144/5 = 28.8

Quantum = 8

Process	Burst Time
P_1	8
P_2	33
P_3	2
P_4	5
P_5	14

SJF: 53/5 = 10.6
RR: 94/5 = 18.8
FCFS: 140/5 = 28.0

Quantum = 8

Process	Burst Time
P_1	9
P_2	33
P_3	2
P_4	5
P_5	14

SJF: 55/5 = 11.0
RR: 111/5 = 22.2
FCFS: 144/5 = 28.8

Quantum = 15

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Queuing Models

- Use queuing theory
- Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process
- Can also generate arrival times for processes (arrival time distribution)

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Queuing Models

- Define a queue for the CPU and a queue for each I/O device and test the various scheduling algorithms
- Knowing the arrival rates and the service rates we can calculate other figures such as average queue length, average wait time, CPU utilization etc.

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Queuing Models

One useful formula is Little's Formula.

$n = \lambda w$

Where

n is the average queue length

λ is the average arrival rate for new processes (e.g. five a second)

w is the average waiting time in the queue

Main disadvantage is that it is not always easy to define realistic distribution times and we have to make assumptions

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Simulations

- A Variable (clock) is incremented
- At each increment the state of the simulation is updated
- Statistics are gathered at each clock tick so that the system performance can be analysed
- Data can be generated in the same way as the queuing model but leads to similar problems

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Simulations

- Use trace data
  - Collected from real processes on real machines
- Disadvantages
  - Simulations can take a long time to run
  - Can take a long time to implement
  - Trace data may be difficult to collect and require large amounts of storage

Scheduling

Processes

Operating Systems

Evaluation of Scheduling Algorithms

Implementation

- Best comparison is to implement the algorithms on real machines
- Best results, but number of disadvantages
  - It is expensive as the algorithm has to be written and then implemented on real hardware
  - If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing
  - If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes