MODERN OPERATING SYSTEMS

Third Edition

ANDREW S. TANENBAUM

Chapter 2
Processes and Threads

outline

- Process
- Threads
- Process scheduling
- Race condition/synchronization

pseudoparallelism

- All modern computers do many things at the same time
- In a uni-processor system, at any instant, CPU is running only one process
- But in a multiprogramming system, CPU switches from processes quickly, running each for tens or hundreds of ms

The Process Model

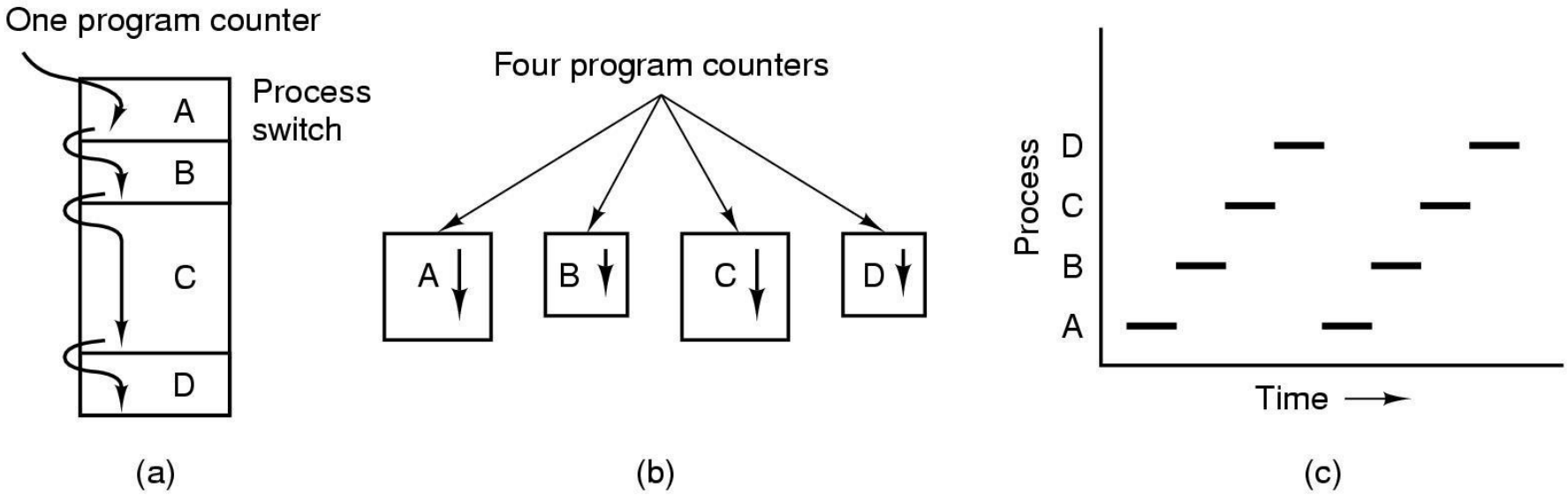


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Advantages and Problems

- The parallelism, though a pseudo one, is very beneficial for users
- However, every coin has two sides…

Execution non-reproducible

- Non-reproducible

Program A
repeat N:=N+1;

Program B
repeat print(N);N:=0;

  - Example at an instant, N＝n，if execution sequence is ：
    - N:＝N＋1； print(N); N:=0; then, N is:n＋1； n＋1； 0
    - print(N); N:=0; N:＝N＋1； then N is: n； 0； 1
    - print(N); N:＝N＋1； N:=0; then N is: n； n＋1； 0

- Another example:
  - An I/O process start a tape streamer and wants to read 1st record
  - It decides to loop 10,000 time to wait for the streamer to speed up

Difference between Process and Program

- Consider an Analogy
- A computer scientist baking a cake for his daughter
- And his son came in crying for help

Process Creation

Events which cause process creation:

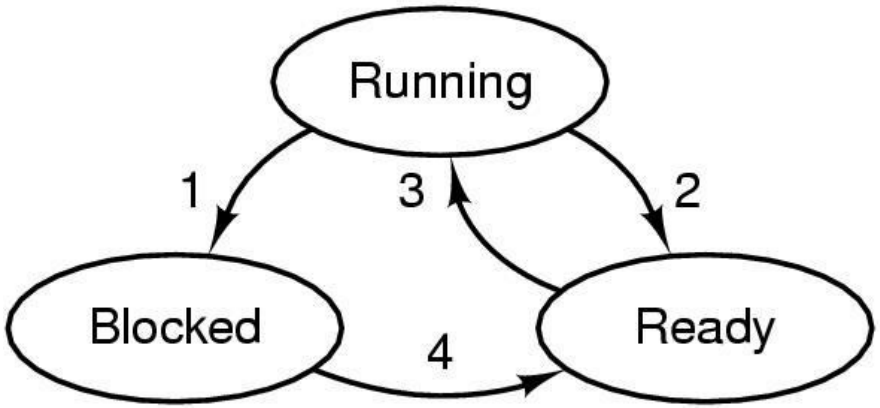
- System initialization.
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

Process Termination

Events which cause process termination:

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Implementation of Processes (1)

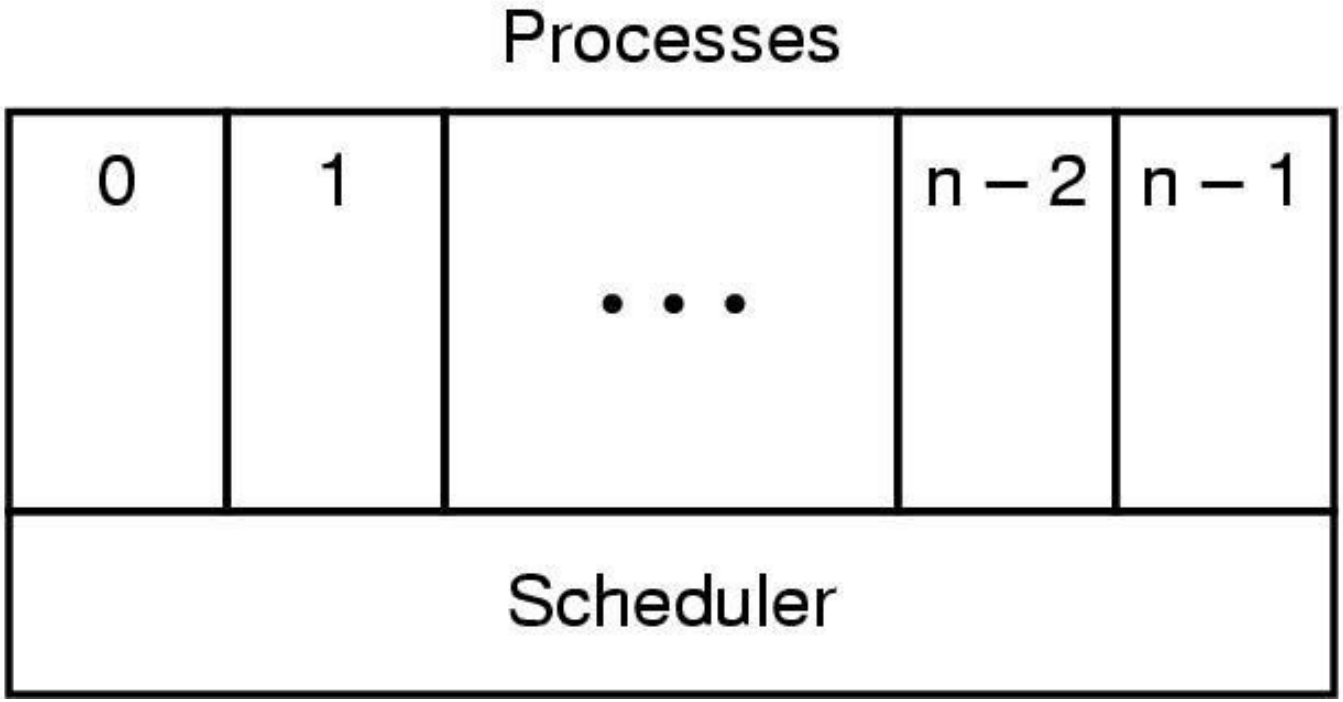


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes (2)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process table entry.

Implementation of Processes (3)

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Modeling Multiprogramming

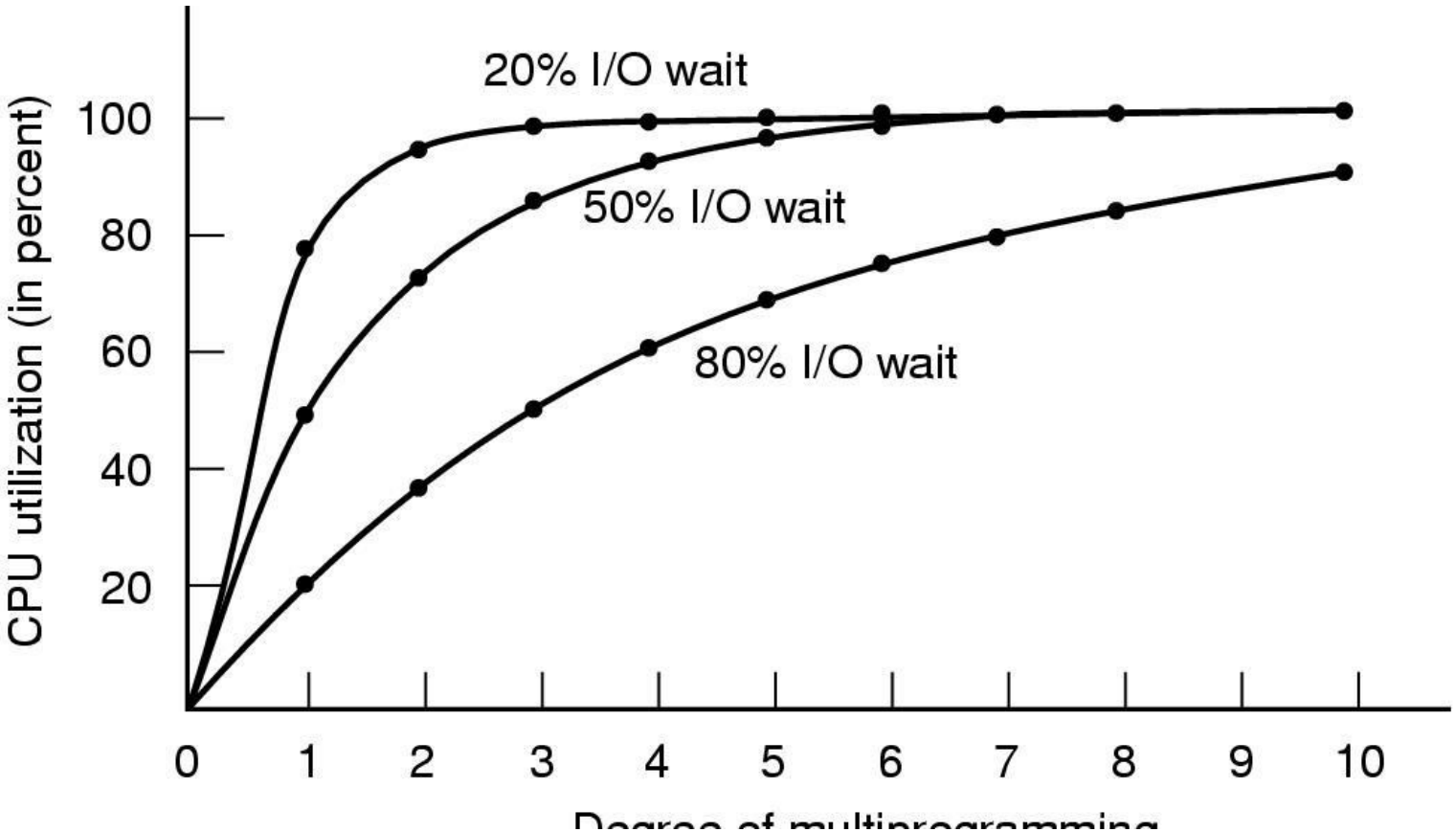
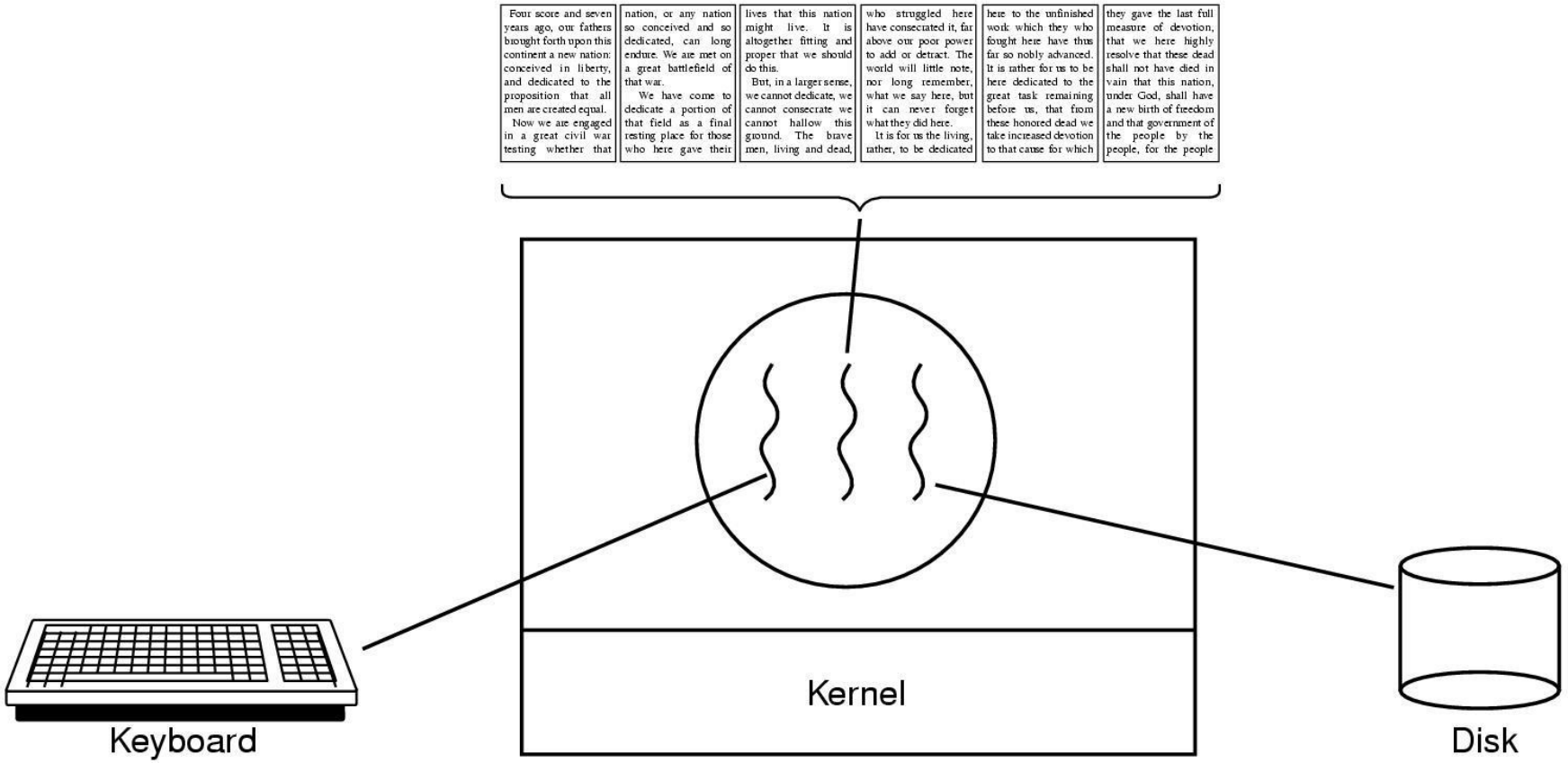


Figure 2-6. CPU utilization as a function of the number of processes in memory.

summary

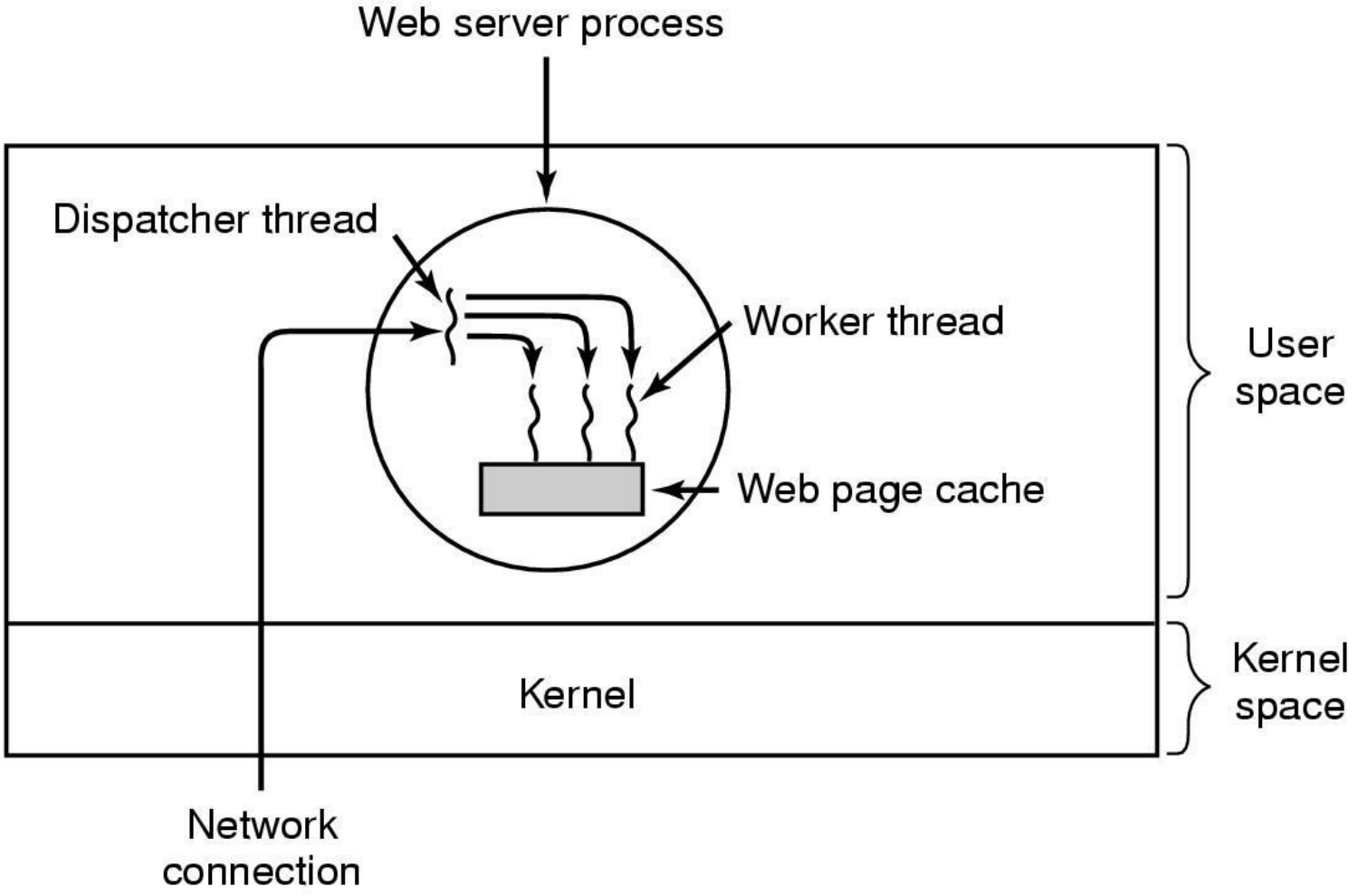
- Pseudoparallelism
- Difference between process and program
- Lifecycle of process
- Multiprogramming degree

Thread Usage (1)



A word processor with three threads

Thread Usage (2)



A multithreaded Web server

Thread Usage (3)

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread

The Thread Model (2)

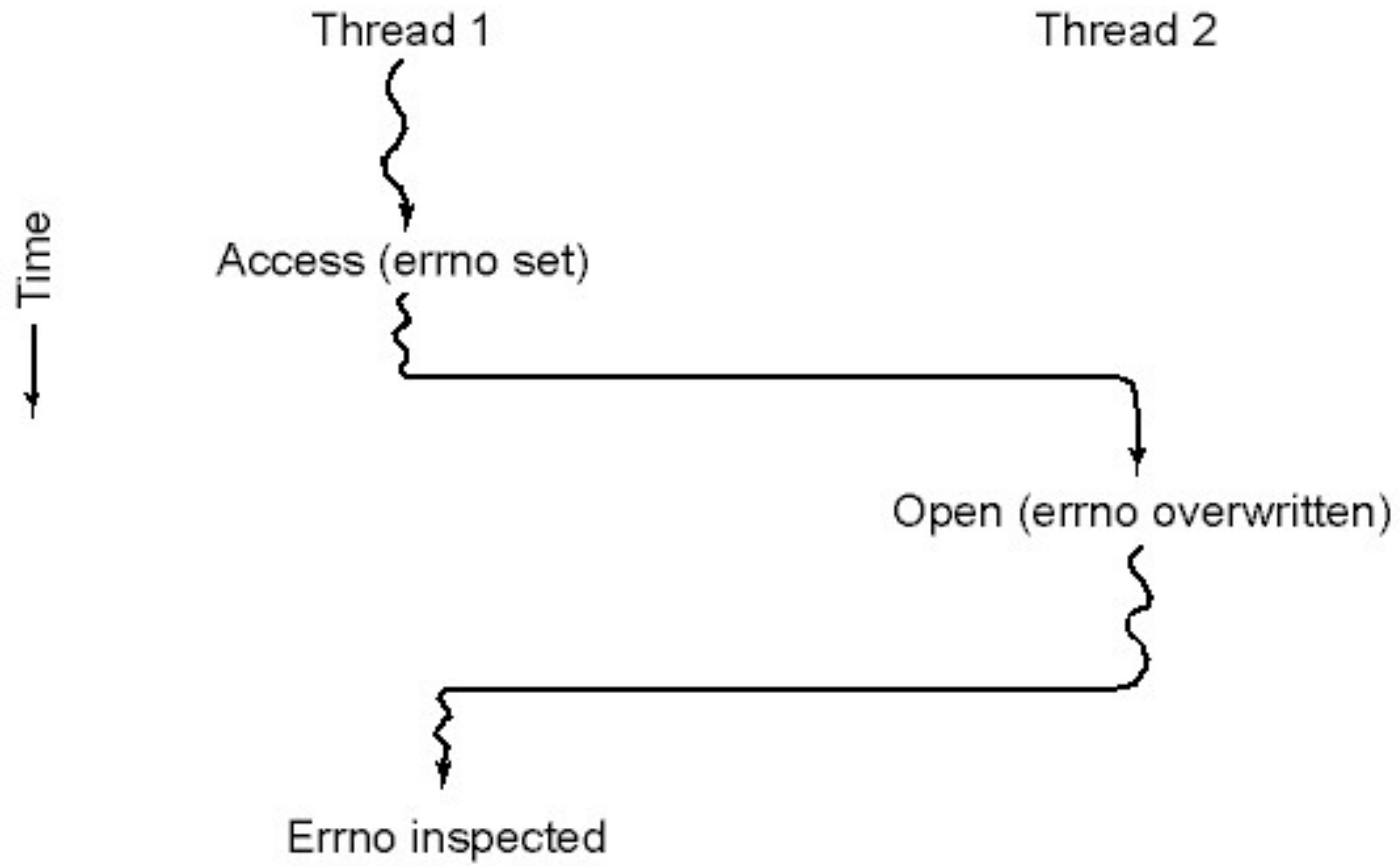
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread

Thread

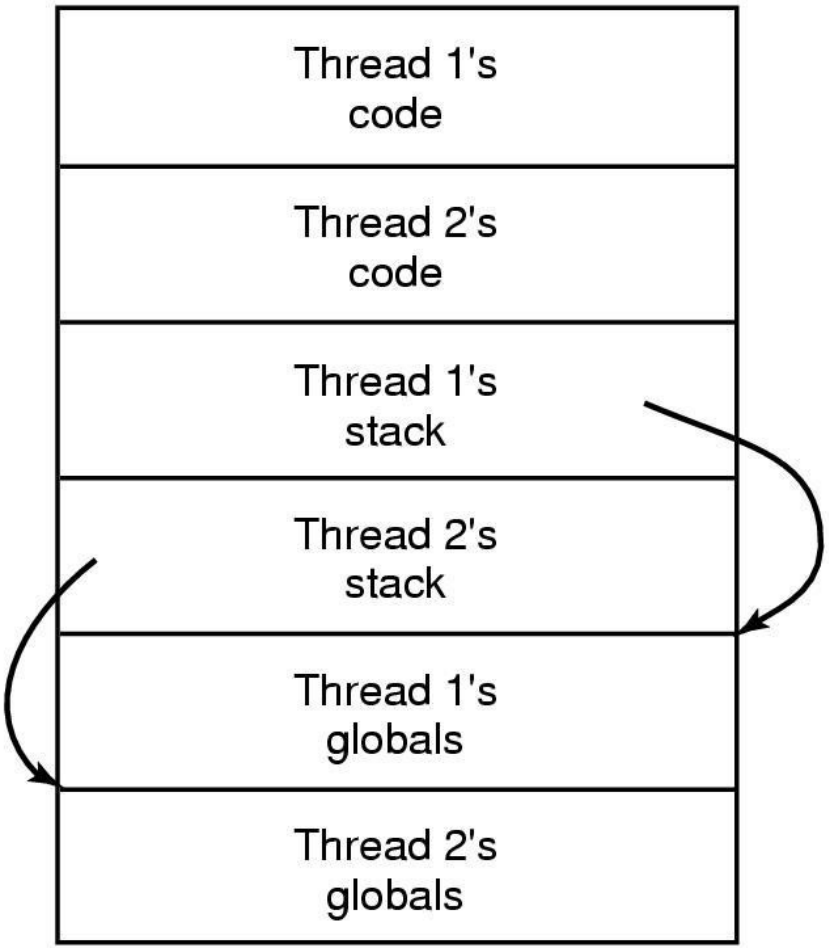
- Has own program counter, register set, stack
- Shares code (text), global data, open files
  - With other threads within a single Heavy-Weight Process (HWP)
  - But not with other HWP's
- May have own PCB
  - Depends on operating system
  - Context will involve thread ID, program counter, register set, stack pointer
  - RAM address space *shared* with other threads in same process
    - Memory management information shared

Making Single-Threaded Code Multithreaded (1)



Conflicts between threads over the use of a global variable

Making Single-Threaded Code Multithreaded (2)



Threads can have private global variables

Threads: Benefits

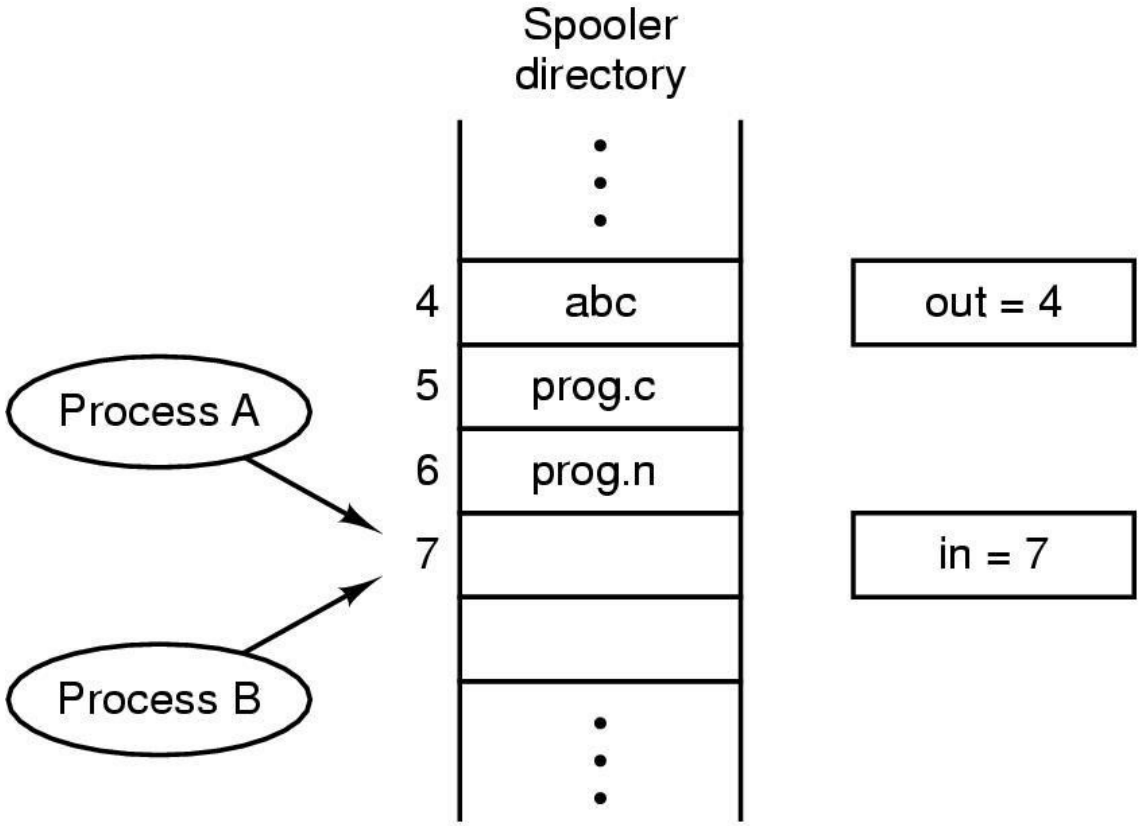
- User responsiveness
  - When one thread blocks, another may handle user I/O
  - But: depends on threading implementation
- Resource sharing: economy
  - Memory is shared (i.e., address space shared)
  - Open files, sockets shared
- Speed
  - E.g., Solaris: thread creation about 30x faster than heavyweight process creation; context switch about 5x faster with thread
- Utilizing hardware parallelism
  - Like heavy weight processes, can also make use of multiprocessor architectures

Threads: Drawbacks

- Synchronization
  - Access to shared memory or shared variables must be controlled ***if the memory or variables are changed***
  - Can add complexity, bugs to program code
  - E.g., need to be very careful to avoid *race conditions, deadlocks* and other problems
- Lack of independence
  - Threads not independent, within a Heavy-Weight Process (HWP)
  - The RAM address space is shared; No memory protection from each other
  - The stacks of each thread are intended to be in separate RAM, but if one thread has a problem (e.g., with pointers or array addressing), it could write over the stack of another thread

Interprocess Communication

Race Conditions



Two processes want to access shared memory at same time

Therac-25

- Between June 1985 and January 1987, some cancer patients being treated with radiation were injured & killed due to faulty software
  - Massive overdoses to 6 patients, killing 3
- Software had a race condition associated with command screen
- Software was improperly synchronized!!
- See also
  - p. 340-341 Quinn (2006), Ethics for the Information Age
  - Nancy G. Leveson, Clark S. Turner, "An Investigation of the Therac-25 Accidents," Computer, vol. 26, no. 7, pp. 18-41, July 1993
    - http://doi.ieeecomputersociety.org/10.1109/MC.1993.274940

Concepts

- Race condition
  - Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
  - In our former example, the possibilities are various

Concetps

- Mutual exclusion
  - Prohibit more than one process from reading and writing the shared data at the same time
- Critical region
  - Part of the program where the shared memory is accessed

Critical Regions (1)

Four conditions to provide mutual exclusion

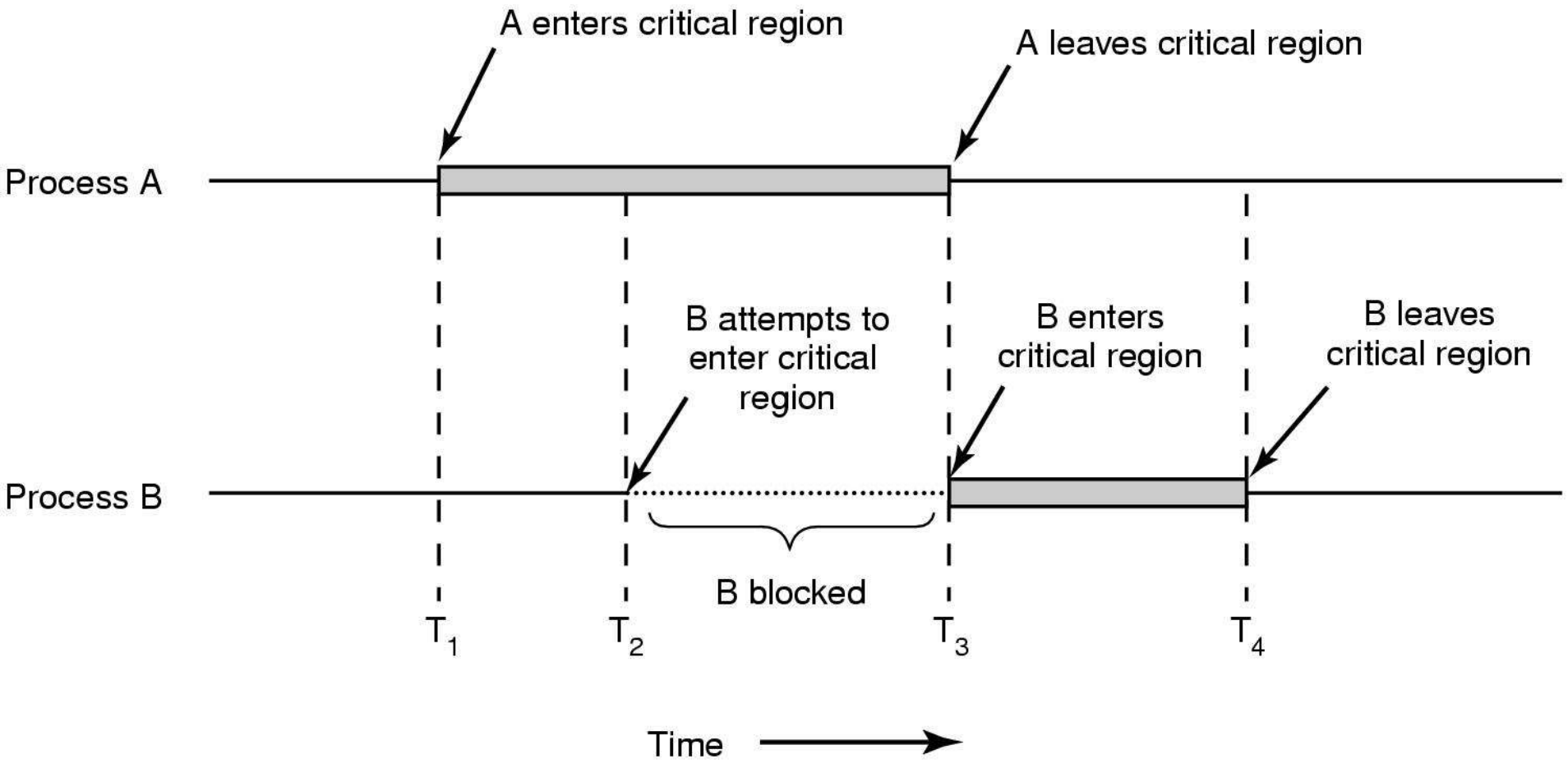
1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Section Problem

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

General structure of a typical process P_i

Critical Regions (2)



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting

- Disable interrupt
  - After entering critical region, disable all interrupts
  - Since clock is just an interrupt, no CPU preemption can occur
  - Disabling interrupt is useful for OS itself, but not for users…

Mutual Exclusion with busy waiting

- Lock variable
  - A software solution
  - A single, shared variable (lock), before entering critical region, programs test the variable, if 0, no contest; if 1, the critical region is occupied
  - What is the problem?
  - An analogy: the notepad on the door…

Mutual Exclusion with Busy Waiting : strict alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed solution to critical region problem

(a) Process 0. (b) Process 1.

Concepts

- Busy waiting
  - Continuously testing a variable until some value appears
- Spin lock
  - A lock using busy waiting is call a spin lock

Mutual Exclusion with Busy Waiting (2) : a workable method

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

Mutual Exclusion with Busy Waiting (3)

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

Entering and leaving a critical region using the TSL instruction

Sleep and wakeup

- Drawback of Busy waiting
  - A lower priority process has entered critical region
  - A higher priority process comes and preempts the lower priority process, it wastes CPU in busy waiting, while the lower priority don't come out
  - Priority inversion/deadlock
- Block instead of busy waiting
  - Wakeup sleep

Producer-consumer problem

- Two processes share a common, fixed-sized buffer
- Producer puts information into the buffer
- Consumer takes information from buffer
- A simple solution

Sleep and Wakeup

```
#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 */
     }
}
```

Producer-consumer problem with fatal race condition

Producer-Consumer Problem

- What can be the problem?

- Signal missing
  - Shared variable: counter
  - Same old problem caused by concurrency
  - When consumer read count with a 0 but didn't fall asleep in time, then the signal will be lost

Semaphore

- Proposed by Dijkstra, introducing a new type of variable
- Atomic Action
  - A single, indivisible action
- Down (P)
  - Check a semaphore to see whether it's 0, if so, sleep; else, decrements the value and go on
- Up (v)
  - Check the semaphore
  - If processes are waiting on the semaphore, OS will chose on to proceed, and complete its **down**
  - **Consider as a sign of number of resources**

Semaphore

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 0 (**binary semaphore**)
  - Synchronization/mutual exclusion

Semaphores

```
#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 */
     }
}
```

The producer-consumer problem using semaphores

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

A problem

- What would happen if the downs in producer's code were reversed in order?

Mutexes in Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads (3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{     int i;

      for (i= 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
            while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
            buffer = i;                         /* put item in buffer */
            pthread_cond_signal(&condc);        /* wake up consumer */
            pthread_mutex_unlock(&the_mutex);/* release access to buffer */
      }
      pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{     int i;

      for (i = 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
            while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
            buffer = 0;                         /* take item out of buffer */
            pthread_cond_signal(&condp);        /* wake up producer */
            pthread_mutex_unlock(&the_mutex);/* release access to buffer */
      }
      pthread_exit(0);
}

int main(int argc, char **argv)
{
      pthread_t pro, con;
      pthread_mutex_init(&the_mutex, 0);
      pthread_cond_init(&condc, 0);
      pthread_cond_init(&condp, 0);
      pthread_create(&con, 0, consumer, 0);
      pthread_create(&pro, 0, producer, 0);
      pthread_join(pro, 0);
      pthread_join(con, 0);
      pthread_cond_destroy(&condc);
      pthread_cond_destroy(&condp);
      pthread_mutex_destroy(&the_mutex);
}
```

. . .

Figure 2-32. Using threads to solve the producer-consumer problem.

Classic IPC Problems

- Dining philosopher problem
  - A philosopher either eat or think
  - If goes hungry, try to get up two forks and eat
- Reader Writer problem
  - Models access to a database

Dining Philosophers Problem (1)

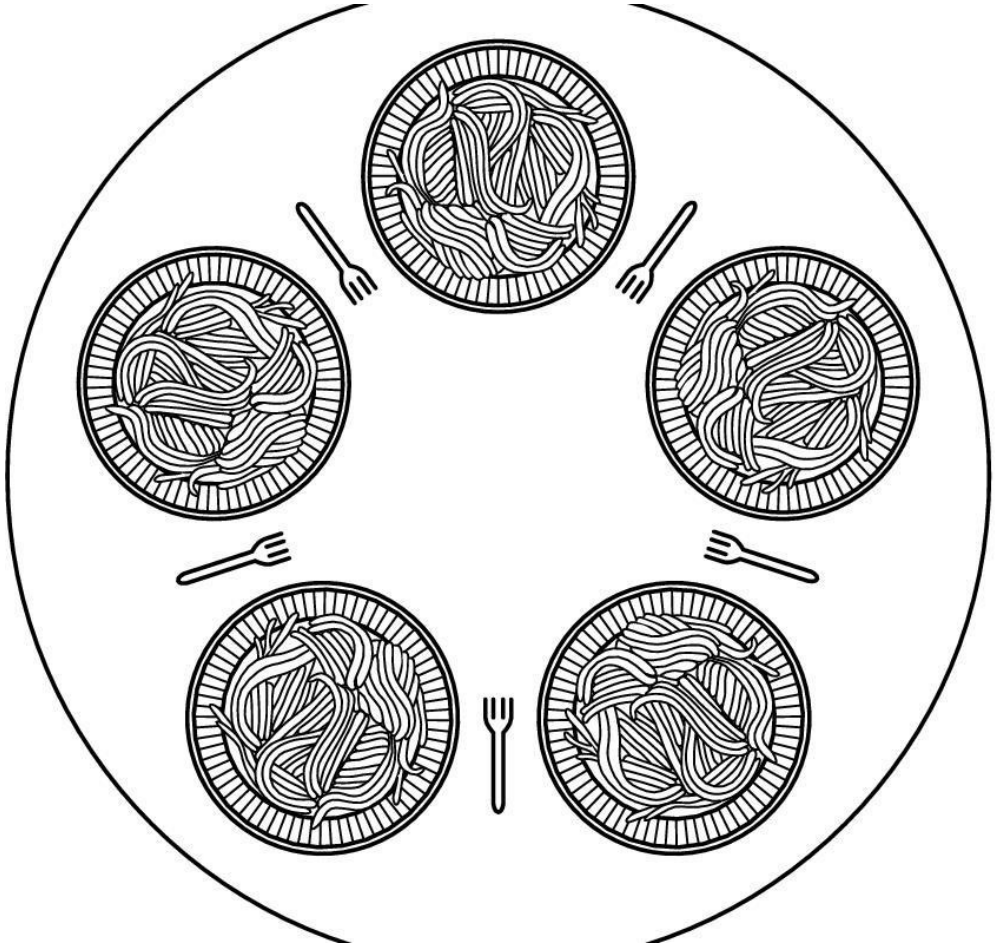


Figure 2-44. Lunch time in the Philosophy Department.

Dining Philosophers Problem (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                           /* philosopher is thinking */
            take_fork(i);                       /* take left fork */
            take_fork((i+1) % N);               /* take right fork; % is modulo operator */
            eat( );                             /* yum-yum, spaghetti */
            put_fork(i);                        /* put left fork back on the table */
            put_fork((i+1) % N);                /* put right fork back on the table */
      }
}
```

Figure 2-45. A nonsolution to the dining philosophers problem.

Dining Philosophers Problem (3)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#defi #define N                5           /* number of philosophers */
#defi #define LEFT             (i+N–1)%N   /* number of i's left neighbor */
#defi #define RIGHT            (i+1)%N     /* number of i's right neighbor */
#defi #define THINKING         0           /* philosopher is thinking */
type( #define HUNGRY           1           /* philosopher is trying to get forks */
int st #define EATING           2           /* philosopher is eating */
sema typedef int semaphore;                 /* semaphores are a special kind of int */
sema int state[N];                          /* array to keep track of everyone's state */
      semaphore mutex = 1;                  /* mutual exclusion for critical regions */
void  semaphore s[N];                       /* one semaphore per philosopher */
{
      void philosopher(int i)               /* i: philosopher number, from 0 to N–1 */
      {
            while (TRUE) {                  /* repeat forever */
                  think( );                 /* philosopher is thinking */
                  take_forks(i);            /* acquire two forks or block */
                  eat( );                   /* yum-yum, spaghetti */
                  put_forks(i);             /* put both forks back on table */
            }
      }
}
```

Figure 2-46. A solution to the dining philosophers problem.

Dining Philosophers Problem (4)

```
. . .
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

. . .
```

Figure 2-46. A solution to the dining philosophers problem.

Dining Philosophers Problem (5)

. . .

```
void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Figure 2-46. A solution to the dining philosophers problem.

The Readers and Writers Problem (1)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
        while (TRUE) {                  /* repeat forever */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(&db); /* if this is the first reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                read_data_base( );      /* access the data */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(&db);   /* if this is the last reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                use_data_read( );       /* noncritical region */
        }
}

  . . .
```

Figure 2-47. A solution to the readers and writers problem.

The Readers and Writers Problem (2)

. . .

```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
                down(&db);              /* get exclusive access */
                write_data_base( );     /* update the data */
                up(&db);                /* release exclusive access */
        }
}
```

Figure 2-47. A solution to the readers and writers problem.

Reader-Writer Problem

- What is the disadvantage of the solution?
  - Writer faces the risk of starvation

scheduling

- Which process to run next?
- When a process is running, should CPU run to its end, or switch between different jobs?

- Process switching is expensive
  - Switch between user mode and kernel mode
  - Current process must be saved
  - Memory map must be saved
  - Cache flushed and reloaded

Scheduling – Process Behavior

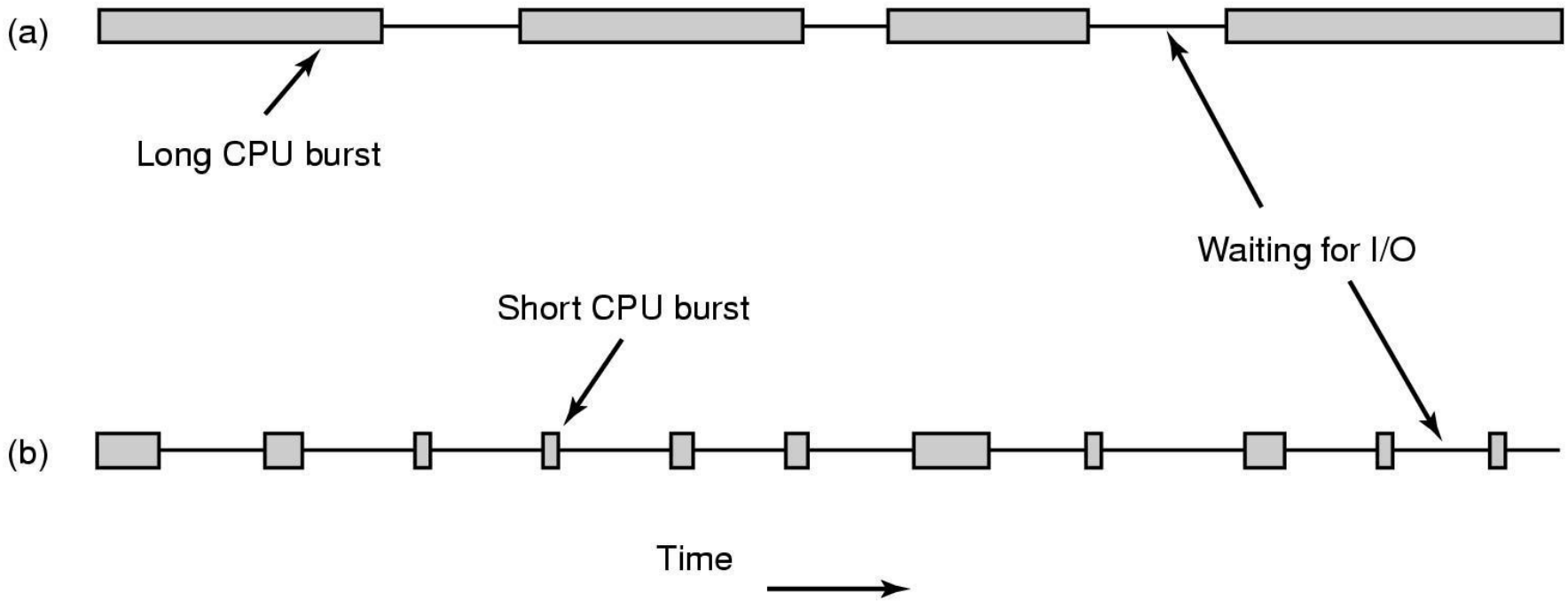


Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Note

- When CPU gets faster, processes are getting more I/O bounded
- A basic idea is if an I/O bound process wants to run, it should get a change quickly

Concepts

- Preemptive algorithm
  - If a process is still running at the end of its time interval, it is suspended and another process is picked up
- Nonpreemptive
  - Picks a process to run and then lets it run till it blocks or it voluntarily releases the CPU

Categories of Scheduling Algorithms

- Different environments need different scheduling algorithms
  - Batch
    - Still in wide use in business world
    - Non-preemptive algorithms reduces process switches
  - Interactive
    - Preemptive is necessary
  - Real time
    - Processes run quickly and block

Scheduling Algorithm Goals

All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Figure 2-39. Some goals of the scheduling algorithm under different circumstances.

Scheduling in Batch Systems

- First-come first-served
- Shortest job first

FCFS

Process	Arrive time	Service time	Start time	Finish time	**Turnaround**	**Weighted turnaround**
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99

- FCFS is advantageous for which kind of jobs?

FCFS

- Disadvantage:
  - One CPU-bound process runs for 1 sec at a time
  - Many I/O bound processes that use little CPU time but each has to perform 1000 disk reads

Shortest Job First

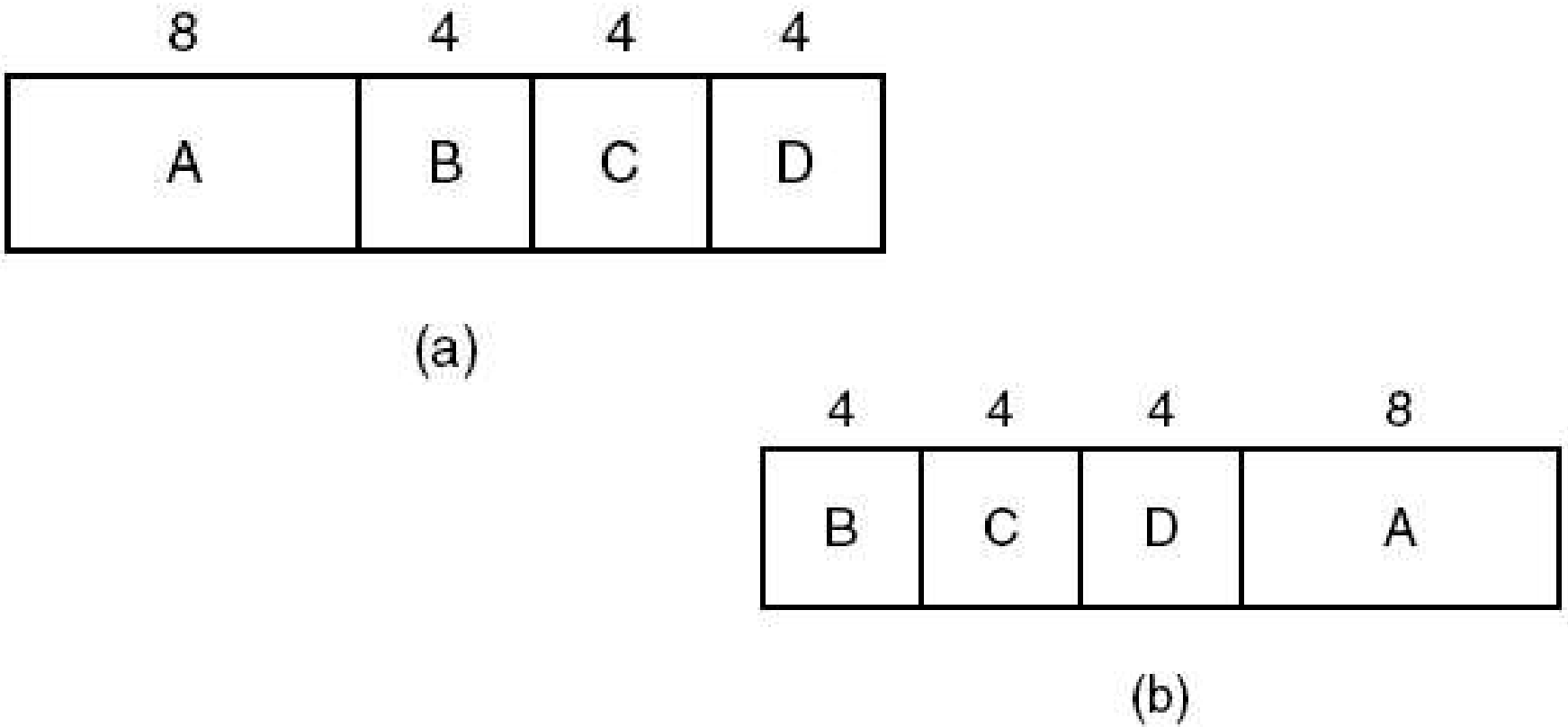


Figure 2-40. An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Shortest Job First

- Turnaround times: shortest job first is provably optimal
- But only when all the jobs are available simultaneously

Shortest job first

	Process	A	B	C	D	E	average
	Arrive time	0	1	2	3	4	
	Service Time	4	3	5	2	4	
SJF	Finish time	4	9	18	6	13	
	turnaround	4	8	16	3	9	8
	weighted	1	2.67	3.1	1.5	2.25	2.1
FCFS	finish	4	7	12	14	18	
	turnaround	4	6	10	11	14	9
	weighted	1	2	2	5.5	3.5	2.8

- Compare:
  - Average Turnaround time
  - Short job
  - Long job

Scheduling in Interactive Systems

- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

Round-Robin Scheduling

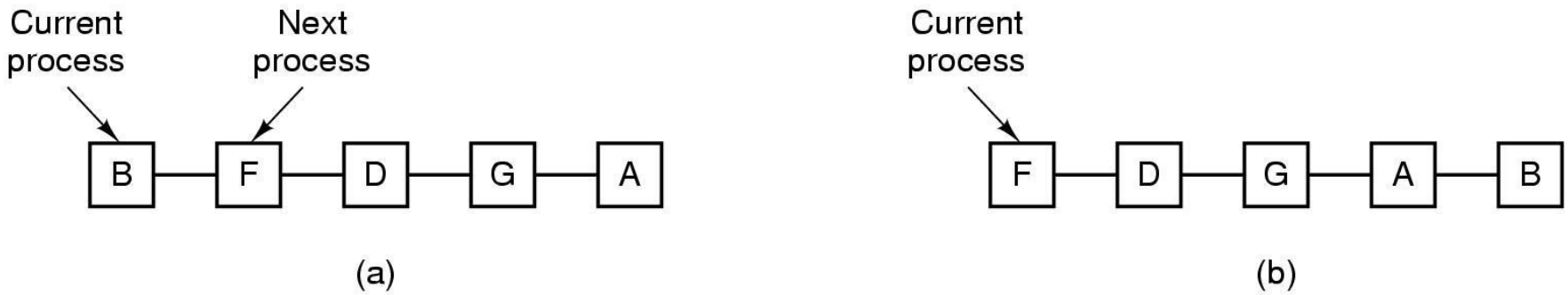


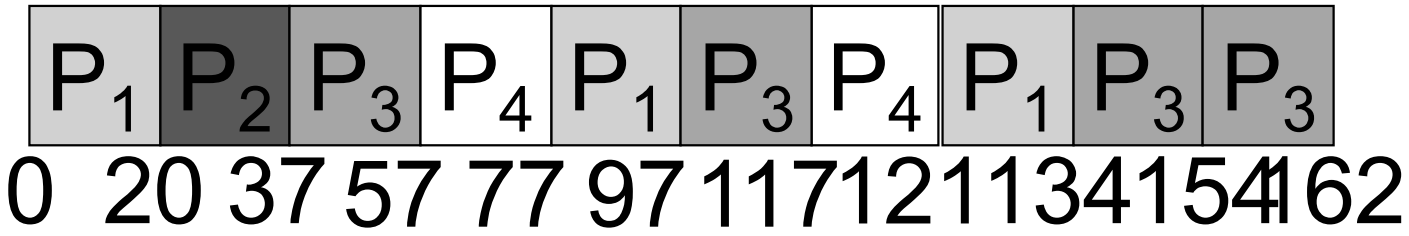
Figure 2-41. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

Example: RR with Time Quantum = 20

- Arrival time = 0
- Time quantum =20

Process	Burst Time
P1	53
P2	17
P3	68
p4	24

- The Gantt chart is



- Typically, higher average turnaround than SJF, but better response

Size of time quantum

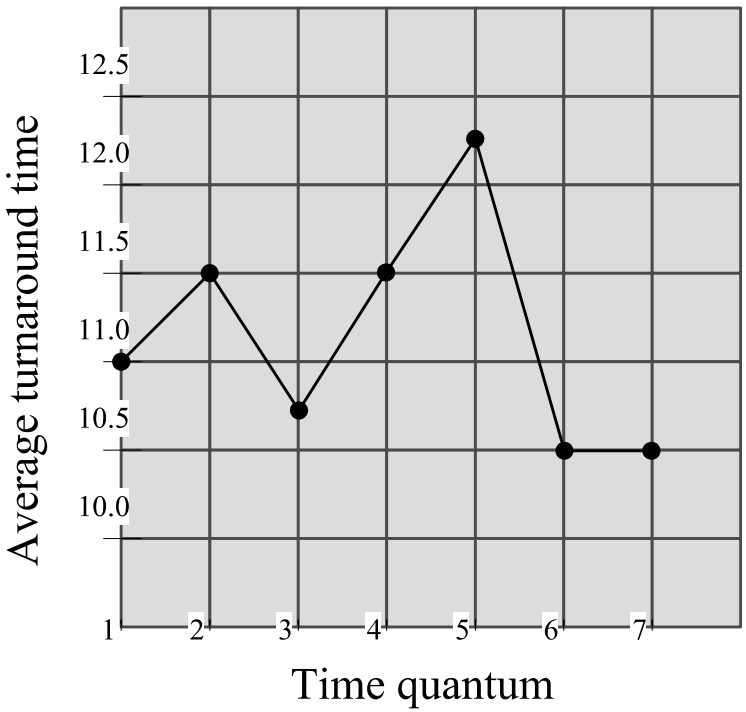
- The performance of RR depends heavily on the size of time quantum
- Time quantum
  - Too big， = FCFS
  - Too small:
    - Hardware: Process sharing
    - Software: context switch, high overhead, low CPU utilization
  - Must be large with respect to context switch

Context switch

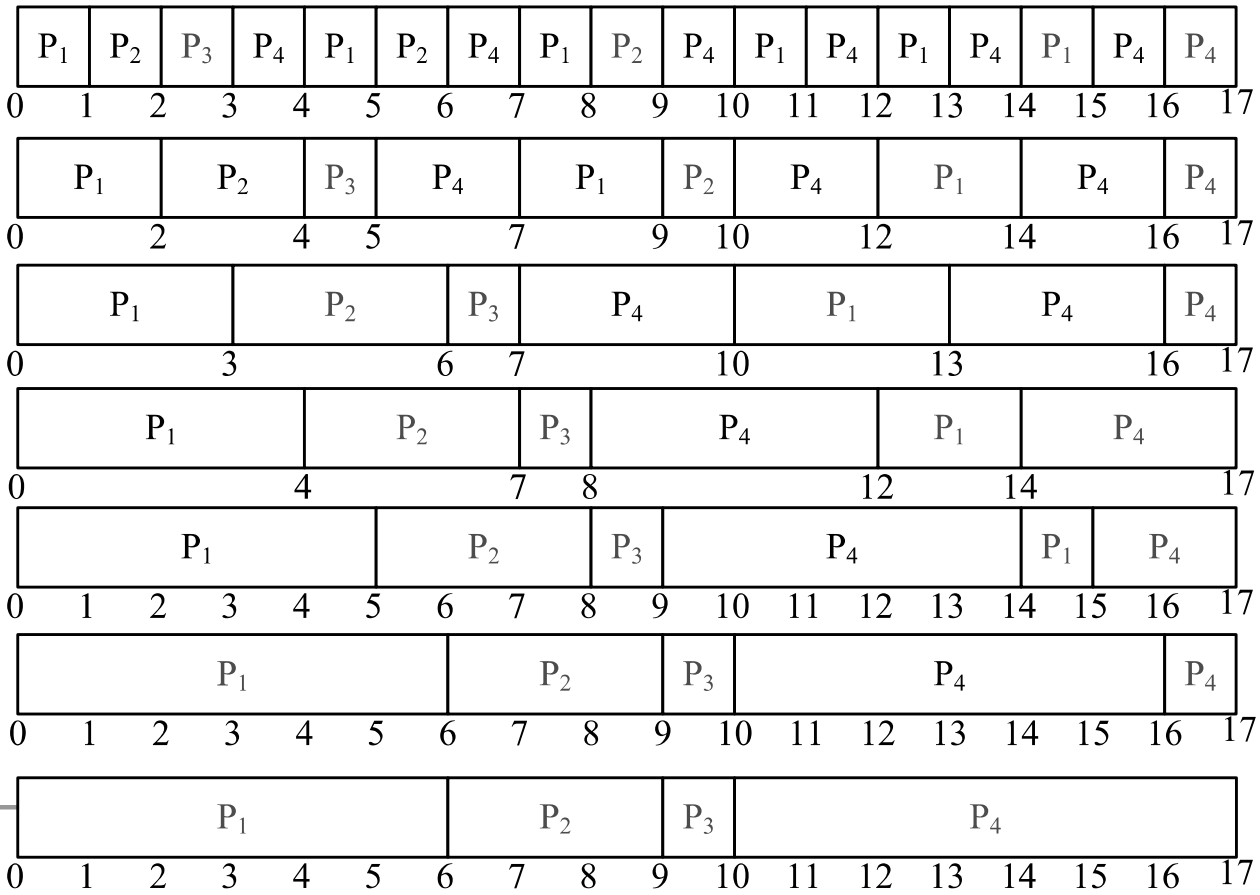
- If the quantum is 4msec, and context switch is 1msec, then 20% of CPU time is wasted
- If the quantum if 100msec, the wasted time is 1%, but less responsive
- A quantum around 20-50 msec is reasonable

Turnaround Time Varies With The Time Quantum

Process	Time
P_1	6
P_2	3
P_3	1
P_4	7



- Will the average turnaround time improve as q increase?
- **80% CPU burst should be shorter than q**



Priority scheduling

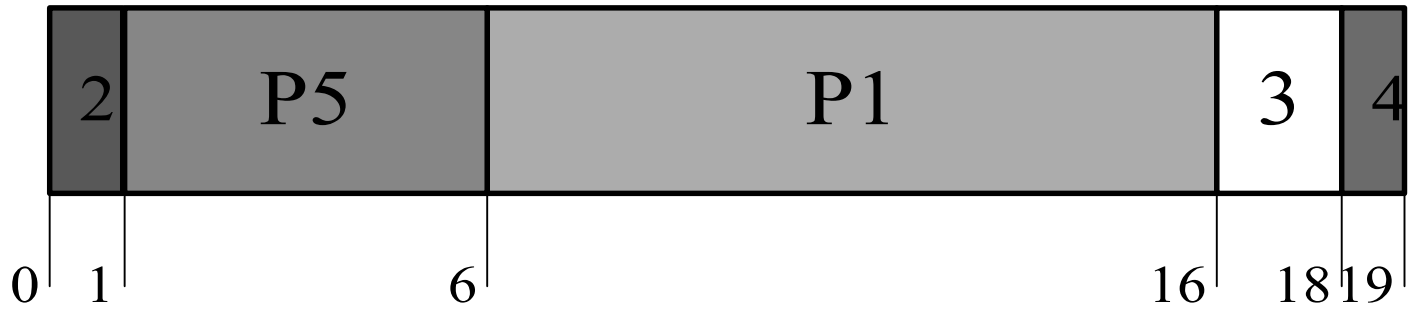
- Each priority has a priority number
- The highest priority can be scheduled first
- If all priorities equal, then it is FCFS

Example

- E.g.:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

- Priority (nonpreemprive)



- Average waiting time

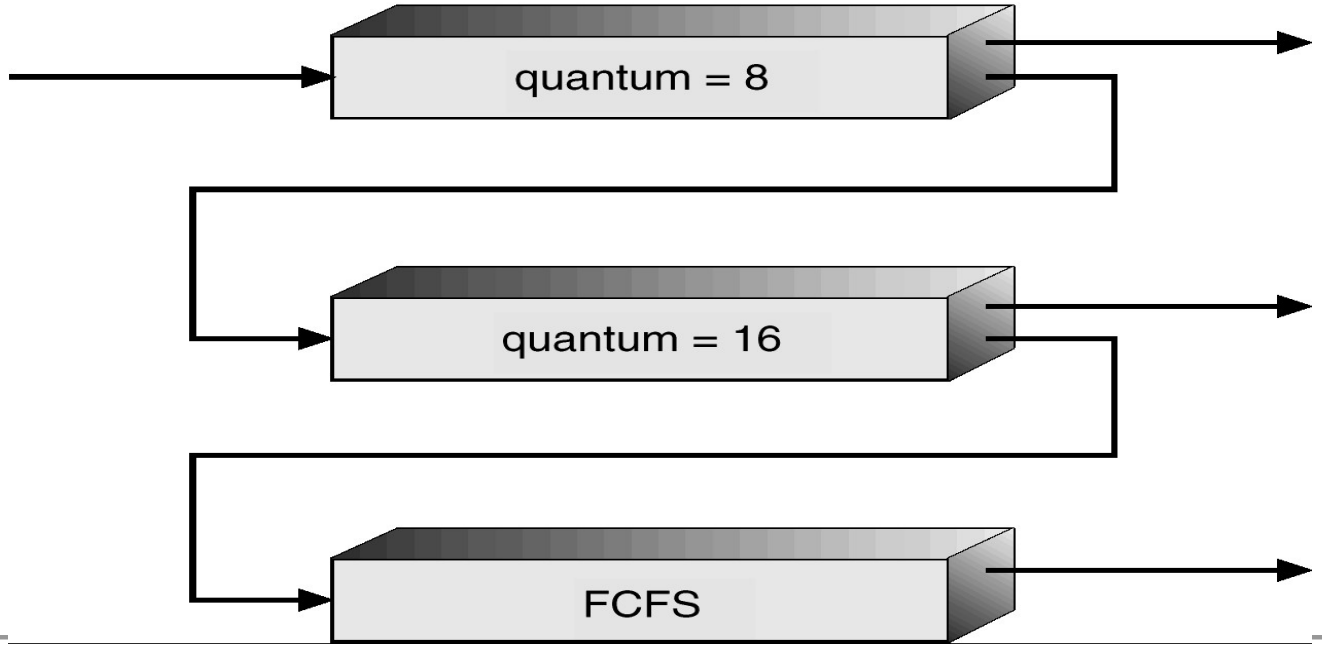
= (6 + 0 + 16 + 18 + 1) /5 = 8.2

Multiple Queues

- CTSS
  - Only one process in memory
  - It is wise to give large quantum to CPU-bound process and small quantum to interactive process
  - Priority classes
    - Highest class run for one quantum; next-highest for two quanta, and so on
    - When a process used up its quantum, it will be moved to the next class

Example of Multiple Queues

- Three queues:
  - Q0 – time quantum 8 milliseconds, FCFS
  - Q1 – time quantum 16 milliseconds, FCFS
  - Q2 – FCFS



Multiple queues

- It is not good for process that firstly CPU-bounded, but later interactive
  - Whenever a carriage return was typed at the terminal, the process belonging to the terminal was moved to the highest priority class
  - What happened?

Shortest Process Next

- In interactive systems, it is hard to predict the remaining time of a process
- Make estimate based on past behavior and run the shortest estimated running time

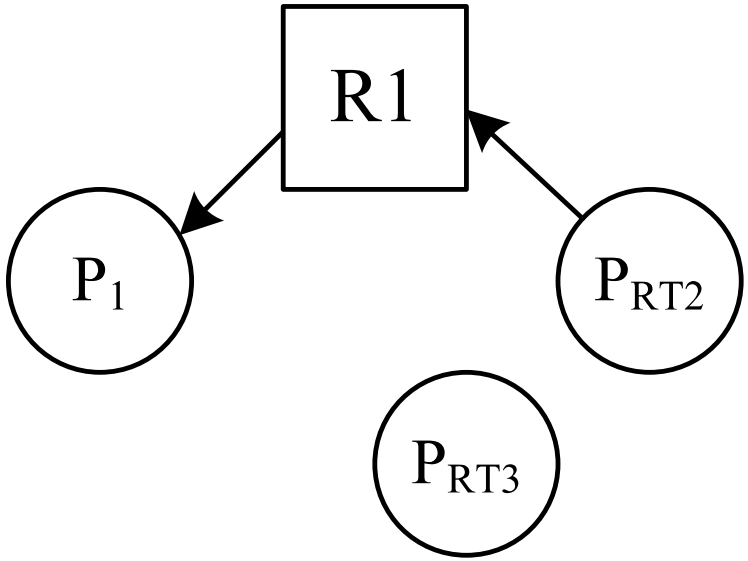
$$T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2 ....$$

- Aging:
  - Estimate next value in a series by taking the weighted averaged of the current and previous estimate

Priority inversion

- Priority inversion
  - **When** the higher-priority process needs to read or modify **kernel data** that are **currently being accessed by** another, **lower-**priority process
  - The **high-**priority process would be **wait**ing **for** a **lower-**priority one to finish

- E.g.:



- Priority: $P_1 < P_{RT3} < P_{RT2}$
- P_{RT3} preempt P_1; P_{RT2} waits for P_1;
- ➔P_{RT2} waits for P_{RT3}

Priority Inversion in Pathfinder

- The landing on Mars of Pathfinder was flawless
- But days later, Pathfinder began to experience series of system resets, casing data loss
- VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Priority Inversion in Pathfinder

- "information bus”
  - a shared memory area used for passing information between different components of the spacecraft.
- bus management task
  - ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).
- meteorological data gathering task ran as an infrequent, low priority thread, used the information bus to publish its data.
- a communications task that ran with medium priority.

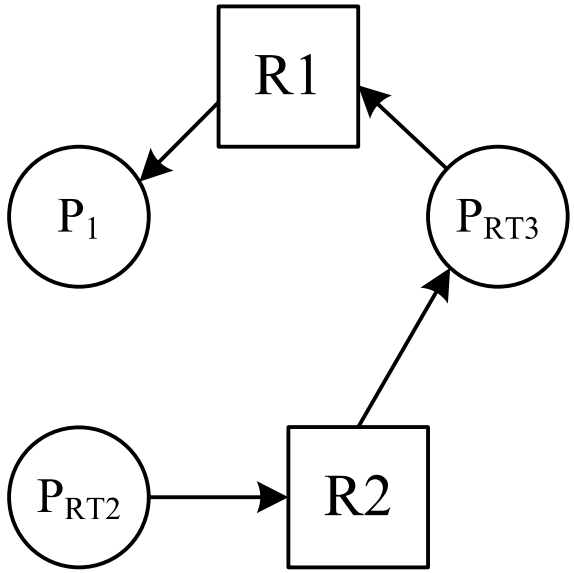
Priority Inversion in Pathfinder

- What would happen if the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread?
- A watchdog would notice that the information bus thread hasn't been executed for some time; and would reset the system

Priority inversion (cont.)

- Solution
  - Priority-inheritance (lending)
    - P_{RT2} lends its priory to P_1, thus P_{RT3} could not preempt P_1

- Priority inheritance must be transitive
  - E.g.:
    - Priority: $P_1 < P_{RT3} < P_{RT2}$



Priority Inversion

- Ceiling Protocol
  - One way to solve priority inversion is to use the *priority ceiling protocol*, which gives each shared resource a predefined priority ceiling.
  - When a task acquires a shared resource, the task is hoisted (has its priority temporarily raised) to the priority ceiling of that resource.
  - The priority ceiling must be higher than the highest priority of all tasks that can access the resource, thereby ensuring that a task owning a shared resource won't be preempted by any other task attempting to access the same resource.
  - When the hoisted task releases the resource, the task is returned to its original priority level.

Exercises

- Compare reading a file using a single-threaded file server and a multithreaded one.
  - It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, if the data are in the cache
  - One-third of the time, it will need disk operation, additional 75 msec is needed
  - How many requests/sec can the server handle if it is single threaded? If it is multithreaded?

Exercises

- Measurements of a certain system have shown that the average process run for a T before blocking on I/O. A process switch requires a time of S. For round-robin scheduling with quantum Q, give the CPU efficiency for each of the following.

$Q = \infty$

$Q > T$

$S < Q < T$

$Q = S$

$Q \to 0$

Exercise

- Five batch jobs A through E, arrive at a computer center at almost the same time, and the estimated running time are 10,6,2,4,and 8 minutes, with priorities 3,5,2,1,4, 5 being the highest. Computer the average turnaround time of the following algorithms.
  - Round robin
  - Priority scheduling
  - FCFS
  - Shortest job first

Exercise

- The aging algorithm with a=1/2 is being used to predict run times. The previous four runs, from oldest to most recent, are 40, 20,40 and 15 msec. What is the prediction of next time?

Exercise

- A process running on CTSS needs 30 quanta to complete. How many times must it be swapped in?