

Processes and Threads

1

Learning Outcomes

- An understanding of fundamental concepts of processes and threads

2

Major Requirements of an Operating System

- Interleave the execution of several processes to maximize processor utilization while providing reasonable response time
- Allocate resources to processes
- Support interprocess communication and user creation of processes

3

Processes and Threads

- Processes:
 - Also called a task or job
 - Execution of an individual program
 - "Owner" of resources allocated for program execution
 - Encompasses one or more threads
- Threads:
 - Unit of execution
 - Can be traced
 - list the sequence of instructions that execute
 - Belongs to a process
 - Executes within it.

4

Execution snapshot of three single-threaded processes (No Virtual Memory)

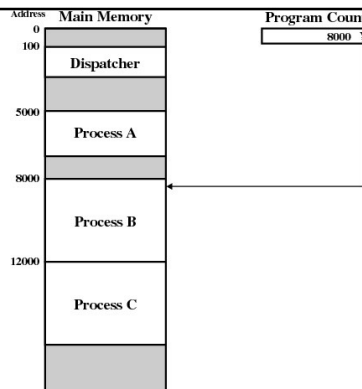


Figure 3.1 Snapshot of Example Execution (Figure 3.1 at Instruction Cycle 13)

5

Logical Execution Trace

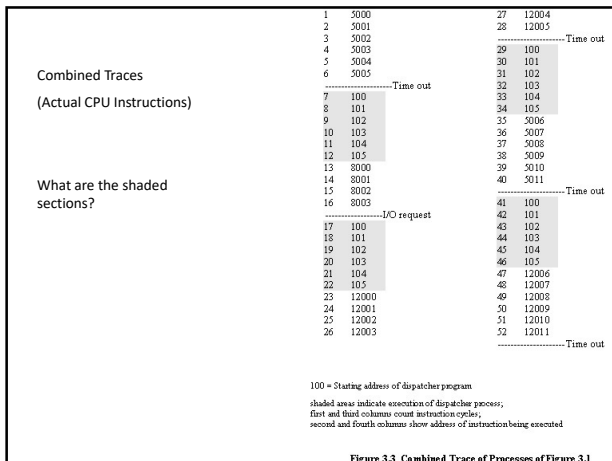
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

5000 = Starting address of program of Process A
 8000 = Starting address of program of Process B
 12000 = Starting address of program of Process C

Figure 3.2 Traces of Processes of Figure 3.1

6

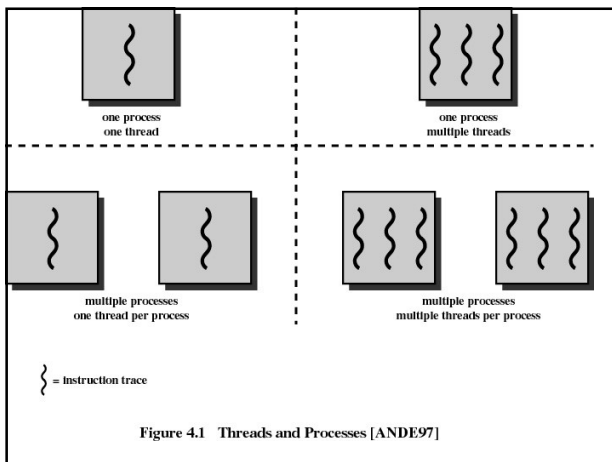


7

Summary: The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes (with a single thread each)
- Only one program active at any instant

8



9

Process and thread models of selected OSes

- Single process, single thread
 - MSDOS
- Single process, multiple threads
 - OS/161 as distributed
- Multiple processes, single thread
 - Traditional UNIX
- Multiple processes, multiple threads
 - Modern Unix (Linux, Solaris), Windows

Note: Literature (incl. Textbooks) often do not cleanly distinguish between processes and threads (for historical reasons)

10

Process Creation

Principal events that cause process creation

1. System initialization
 - Foreground processes (interactive programs)
 - Background processes
 - Email server, web server, print server, etc.
 - Called a *daemon* (unix) or *service* (Windows)
2. Execution of a process creation system call by a running process
 - New login shell for an incoming ssh connection
3. User request to create a new process
4. Initiation of a batch job

Note: Technically, all these cases use the same system mechanism to create new processes.

11

Process Termination

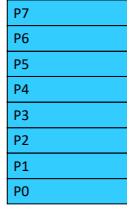
Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

12

Implementation of Processes

- A processes' information is stored in a *process control block* (PCB)
- The PCBs form a *process table*
 - Reality can be more complex (hashing, chaining, allocation bitmaps,...)



13 UNSW

13

Implementation of Processes

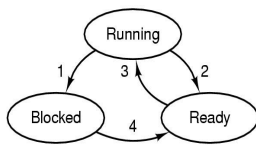
Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters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

Example fields of a process table entry

14 UNSW

14

Process/Thread States



- Process blocks for input
- Scheduler picks another process
- Scheduler picks this process
- Input becomes available

- Possible process/thread states
 - running
 - blocked
 - ready
- Transitions between states shown

15 UNSW

15

Some Transition Causing Events

Running → Ready

- Voluntary `yield()`
- End of timeslice

Running → Blocked

- Waiting for input
 - File, network,
- Waiting for a timer (alarm signal)
- Waiting for a resource to become available

16 UNSW

16

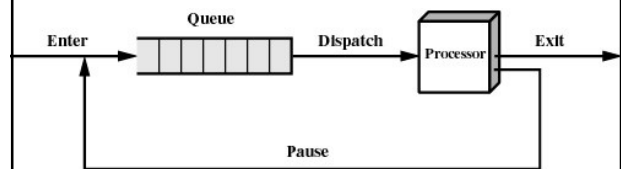
Scheduler

- Sometimes also called the *dispatcher*
 - The literature is also a little inconsistent on with terminology.
- Has to choose a *Ready* process to run
 - How?
 - It is inefficient to search through all processes

17 UNSW

17

The Ready Queue



(b) Queuing diagram

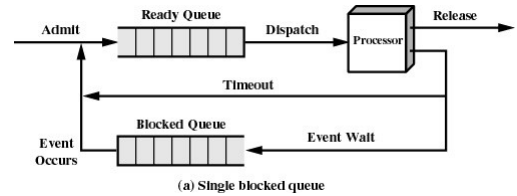
18 UNSW

18

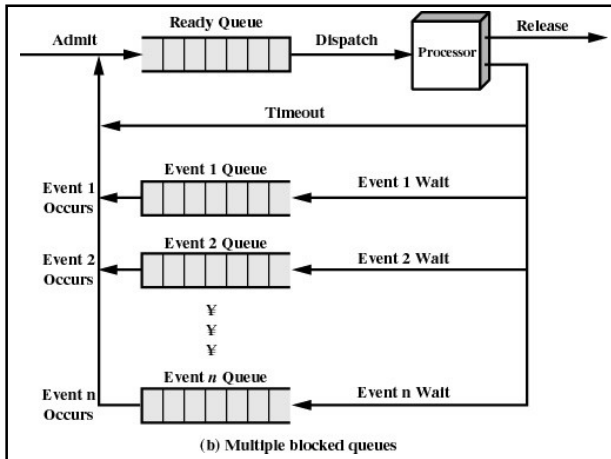
What about blocked processes?

- When an *unblocking* event occurs, we also wish to avoid scanning all processes to select one to make *Ready*

Using Two Queues

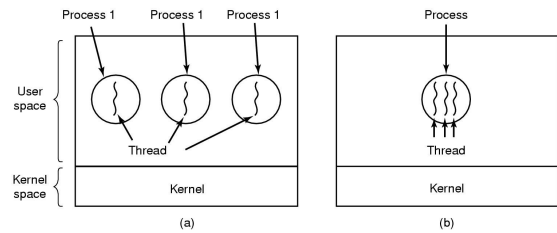


(a) Single blocked queue



(b) Multiple blocked queues

Threads The Thread Model



- (a) Three processes each with one thread
- (b) One process with three threads

The Thread Model – Separating execution from the environment.

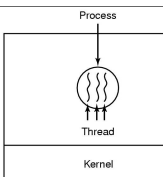
Per process items

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

Per thread items

- Program counter
- Registers
- Stack
- State

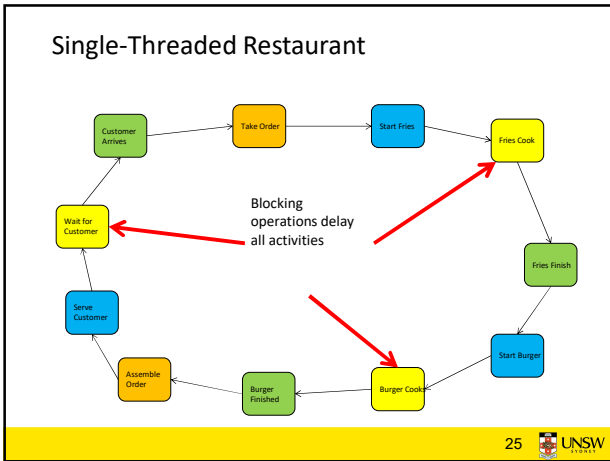
- Items shared by all threads in a process
- Items private to each thread



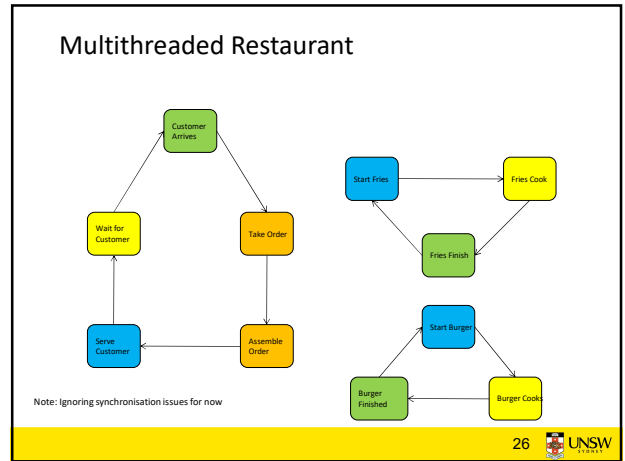
Threads Analogy



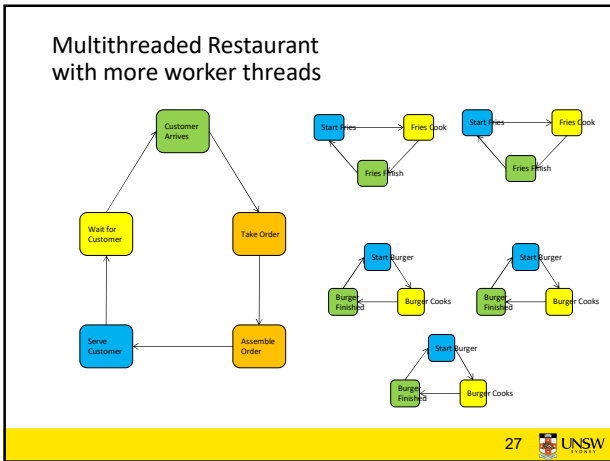
The Hamburger Restaurant



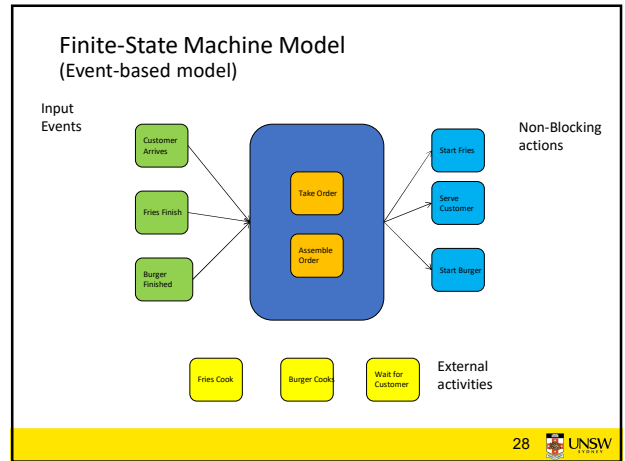
25



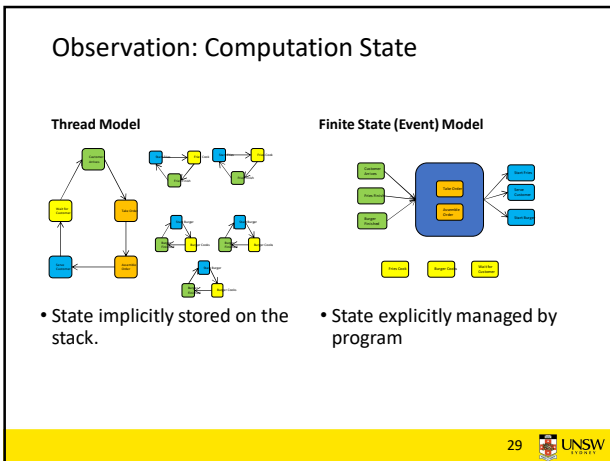
26



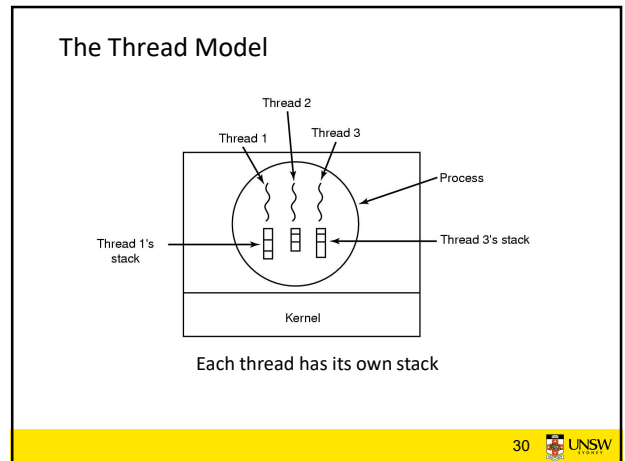
27



28



29

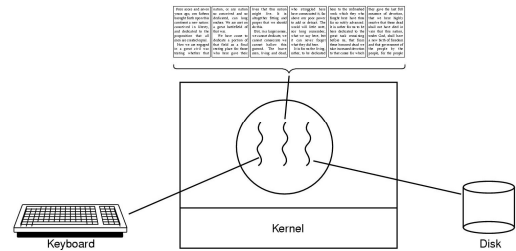


30

Thread Model

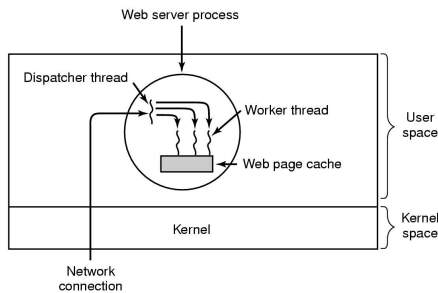
- Local variables are per thread
 - Allocated on the stack
- Global variables are shared between all threads
 - Allocated in data section
 - Concurrency control is an issue
- Dynamically allocated memory (malloc) can be global or local
 - Program defined (the pointer can be global or local)

Thread Usage



A word processor with three threads

Thread Usage



A multithreaded Web server

Thread Usage

```

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 – can overlap disk I/O with execution of other threads

Thread Usage

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

Summarising “Why Threads?”

- Simpler to program than a state machine
- Less resources are associated with them than a complete process
 - Cheaper to create and destroy
 - Shares resources (especially memory) between them
- Performance: Threads waiting for I/O can be overlapped with computing threads
 - Note if all threads are *compute bound*, then there is no performance improvement (on a uniprocessor)
- Threads can take advantage of the parallelism available on machines with more than one CPU (multiprocessor)